DPF: A Data Parallel Fortran Benchmark Suite

Citation

Hu, Yu, S. Lennart Johnsson, Dimitris Kehagias, and Nadia Shalaby. 1995. DPF: A Data Parallel Fortran Benchmark Suite. Harvard Computer Science Group Technical Report TR-36-95.

Permanent link

<http://nrs.harvard.edu/urn-3:HUL.InstRepos:29323487>

Terms of Use

This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at <http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA>

Share Your Story

The Harvard community has made this article openly available.
Please share how this access benefits you. [Submit a story](#).

[Accessibility](#)

DPF: A Data Parallel Fortran Benchmark Suite

Yu Hu
S. Lennart Johnsson
Dimitris Kehagias
Nadia Shalaby

TR-36-95
September 1995

Center for Research in Computing Technology
Harvard University
Cambridge, Massachusetts

DPF: A Data Parallel Fortran Benchmark Suite[†]

Yu Hu[†], S. Lennart Johnsson^{‡§}, Dimitris Kehagias[†] and Nadia Shalaby[†]

November 10, 1996

Abstract

The Data Parallel Fortran (DPF) benchmark suite is designed for evaluating data parallel compilers and scalable architectures. Many of the DPF codes are provided in three versions: basic, optimized and with library calls for performance critical operations typically found in software libraries. The functionality of the benchmarks cover collective communication functions, scientific software library functions, and application kernels that reflect the computational structure and communication patterns in fluid dynamic simulations, fundamental physics and molecular studies in chemistry and biology. The intended target language is High Performance Fortran (HPF). However, due to the lack of HPF compilers at the time of this benchmark development, the suite is written in Connection Machine Fortran (CMF). The DPF codes provide performance evaluation metrics in the form of busy and elapsed times (CM-5), FLOP rates and arithmetic efficiency. The codes are characterized in terms of FLOP count, memory usage, communication pattern, local memory accesses, array allocation mechanism, as well as operation and communication counts per iteration.

[†]The DPF benchmark suite of programs was developed by Thinking Machines Corp. with partial support from ARPA under subcontract DABT63-91-C-0031 with the Computer Science Department of Yale University and the Northeast Parallel Architectures Center (NPAC) at Syracuse University. Verification, debugging and documentation were made in part by the Parallel Computation Research Group at Harvard University with support from Thinking Machines Corp.

[‡]Harvard University, Cambridge, MA 02138

[§]University of Houston, Houston, TX 77204

Contents

1	Introduction	1
1.1	Motivation, Functionality and Scope	1
1.2	Architectural Model	3
1.3	Language Aspects	4
1.4	DPF Code Versions	6
1.5	Performance Evaluation	8
1.6	Benchmark Guide	9
2	Library Functions for Communication	11
2.1	Gather	12
2.2	Scatter	13
2.3	Reduction	13
2.4	Transpose	14
3	Library Functions for Linear Algebra	14
3.1	Matrix–Vector Multiplication	15
3.2	LU Factorization and Solution	18
3.3	QR Factorization and Solution	18
3.4	Gauss–Jordan Matrix Inversion	19
3.5	Tridiagonal Solver	20
3.5.1	Parallel Cyclic Reduction	20
3.5.2	Conjugate Gradient	22
3.6	Jacobi Eigenanalysis	22
3.7	Fast Fourier Transform	24
4	Application Kernels	25
4.1	Boson: many–body simulation	26
4.2	Diffusion equation	29
4.2.1	1–D Diffusion Equation: Tridiagonal Solver	31
4.2.2	2–D Diffusion Equation: ADI Method	33
4.2.3	3–D Diffusion Equation: Explicit Finite Difference	35
4.3	Solution of Poisson’s equation by the Conjugate Gradient method	36
4.4	Finite element method in three dimensions	37
4.5	Fermion	38

4.6	Seismic processing	39
4.7	Spectral method: integration of Kuramoto–Sivashiniski equations	40
4.8	Molecular dynamics, Lennard–Jones force law	41
4.8.1	Local forces only	41
4.8.2	Long range forces	43
4.9	A generic direct N–body solver, long range forces	45
4.10	Particle–in–cell codes	46
4.10.1	Straightforward implementation	46
4.10.2	Sophisticated implementation	47
4.11	QCD kernel: staggered fermion Conjugate Gradient method	49
4.12	Quantum Monte–Carlo	51
4.13	Quadratic programming problem	53
4.14	Solution of nonsymmetric linear equations using the Conjugate Gradient method	54
4.15	Explicit finite difference in two dimensions	55
4.16	Wave equation	56
5	Summary	57

List of Tables

1	Summary of Fortran-90 and HPF language constructs in the DPF benchmark codes . . .	5
2	The code versions in the overall benchmark suite	7
3	Number of FLOPs for each operation type	8
4	Data type sizes (in bytes) and symbols for standard 32-bit based arithmetic architectures	9
5	Local memory access mechanisms and types of array allocation for the communication codes	11
6	Data representation and layout for dominating computations in linear algebra kernels (“:serial” represents a local axis, “:” represents a parallel axis)	15
7	Communication pattern of linear algebra kernels (the array dimensions for reduction and broadcast are of source and destination respectively)	16
8	Computation to communication ratio and memory usage in linear algebra kernels	16
9	Local memory access mechanisms and types of array allocation for the linear algebra codes	17
10	FLOP count for QR factorization and solution	19
11	Memory usage for QR factorization and solution	19
12	Memory usage for parallel cyclic reduction on tridiagonal systems	22
13	FLOP count and memory usage for FFT	25
14	Characterization of the application kernels	27
15	Data representation and layout of arrays for dominating computations in the application kernels (“:serial” represents a local axis, “:” represents a parallel axis)	27
16	Communication patterns in application kernels	28
17	Implementation techniques for some common communication patterns in application kernels	29
18	Computation to communication ratio and memory usage in main loop of application kernels	30
19	Local memory access mechanisms and types of array allocation for the application kernels	31
20	FLOP count and memory usage for vortex simulations	46

1 Introduction

1.1 Motivation, Functionality and Scope

The goal in developing the Data Parallel Fortran (DPF) benchmark suite was to produce a means for evaluating data parallel Fortran compilers, such as the emerging High Performance Fortran (HPF) [18] compilers, Fortran-90 [49] compilers, the Fortran-Y [9] compiler ¹, as well as the Connection Machine Fortran (CMF) [64] compiler from Thinking Machines Corp. At the time the benchmarks were developed, CMF was the only data parallel Fortran language with a production quality compiler available. Hence, the benchmarks were all written in CMF. Conversion to any of the above Fortran versions, in particular HPF, is fairly straight-forward given the limited differences between CMF and HPF. Conversion to Fortran-90 either directly or via HPF should also be fairly straightforward.

The functionality of the benchmarks cover collective communication functions, scientific software library functions, and application kernels. High performance is the main motivation for scalable architectures, while portability of user codes is critical for all but a few applications in making scalable architectures economically feasible. These requirements represent a significant challenge for all software developers, whether they are developing compilers, run-time systems, operating systems or software libraries. One motivation for libraries, in particular in the early years of new architectures, is that they may offer significantly higher performance by being implemented, at least in part, in lower level languages to avoid deficiencies in compiler technology, or in the implementation of compilers and run-time systems. However, though the functionality of libraries is limited compared to that of applications being run on most computers, implementing libraries in low level languages tend to be very costly, and often means that high or even good performance may not be available until late in the hardware product cycle. This in turn implies that following the rapid advances in hardware technology is very difficult, since the older generation hardware often competes successfully with the new generation because of the difference in the quality of software. Thus, it is important to minimize the amount of low level code also in software libraries.

That data motion in memory hierarchies is critical for high performance is a well known fact. Efficient exploitation of spatial and temporal locality of reference is the main objective of compilers for high performance. Some functions require efficient run-time system support; gather and scatter operations are such examples. For conventional vector architectures, gather and scatter operations have been implemented as special instructions, and array transposition has been included in some languages as an intrinsic function, as for instance in Fortran-90. Reduction and broadcast operations are other examples of operations on collection of variables that are incorporated in modern languages with Fortran-90 again being an example. On scalable architectures, the functions mentioned above are usually implemented as part of a collective communication library, which may be part of the run-time system or be a separate library. Several of the functions mentioned are incorporated in the emerging Message-Passing Interface (MPI) standard [48]. The DPF benchmark suite contains codes that either implement this functionality in CMF or calls its supporting Run-Time System.

In addition to some of the most common collective communication functions and functions that are frequently occurring in many basic scientific software libraries, the DPF benchmark suite also contains a set of small application codes containing typical “inner loop” constructs that are

¹Later carried out at Cooperating Systems Corp. as the CRAFT compiler

critical for performance, but that are typically not found in libraries. An example is stencil evaluations in explicit finite difference codes. The benchmarks were chosen to complement each other, such that a good coverage would be obtained of language constructs and idioms frequently used in scientific applications, and for which high performance is critical for good performance of the entire application. Much of the resources at supercomputer centers are consumed by codes used in fluid dynamic simulations, in fundamental physics and in molecular studies in chemistry or biology. The selection of application codes in the DPF benchmark suite reflects this fact.

The objective of the DPF benchmark suite is to serve as an instrument in the evaluation of data parallel compilers and scalable architectures. The NAS kernels [4] and the Livermore loops [47] are examples of “classical” benchmarks designed for uniprocessors. The NAS parallel benchmarks [2], are “paper and pencil” benchmarks aimed at evaluating systems. The benchmarks specify the task to be performed and allows the implementor to choose algorithms as well as programming model. The NAS parallel benchmarks 2.0 [3] consists of MPI-based source implementations. The Perfect Club benchmarks [16, 11, 61] is a collection of Fortran 77 application codes for the evaluation of sequential architectures. The Parkbench project [25] has assembled a collection of sequential and parallel benchmarks for the message passing programming paradigm. To our knowledge, the DPF suite is the first benchmark suite focused entirely on data parallel software environments.

The DPF benchmark suite is organized into two main groups: *library functions* and *application kernels*. The names by which we refer to the codes are given in parenthesis.

Library functions are of two types:

Communication functions

1. *Gather* (**gather**)
2. *Scatter* (**scatter**)
3. *Reduction* (**reduction**)
4. *Transpose* (**transpose**)

Linear algebra functions

1. *Matrix-Vector multiplication* (**matrix-vector**)
2. *LU factorization and solution* (**lu-factor**, **lu-solve**)
3. *QR factorization and solution* (**qr-factor**, **qr-solve**)
4. *Gauss-Jordan matrix inversion* (**gauss-jordan**)
5. *Triangular solvers*:
 - (a) *Parallel cyclic reduction* (**pcr**)
 - (b) *Conjugate Gradient* (**conj-grad**)
6. *Jacobi eigenanalysis* (**jacobi**)
7. *Fast Fourier transform* (**fft**)

Applications kernels cover the following applications and methods:

1. *Boson*: many-body simulation (**boson**)
2. *Diffusion equation*:
 - (a) one-dimensional using tridiagonal solvers (**diff-1D**)
 - (b) two-dimensional using the ADI method (**diff-2D**)

- (c) three-dimensional using an explicit finite difference algorithm (`diff-3D`)
- 3. *Poisson's equation* by the Conjugate Gradient method (`ellip-2D`)
- 4. *Solution of the equilibrium equations in three dimensions by the Finite Element Method* (`fem-3D`)
- 5. *Fermion*: quantum many-body computation for fermions on a lattice (`fermion`)
- 6. *Seismic processing*: generalized moveout (`gmo`)
- 7. *Spectral method*: integration of Kuramoto–Sivashinski equations (`ks-spectral`)
- 8. *Molecular dynamics, Leonard–Jones force law*:
 - (a) with local forces only (`mdcell`)
 - (b) with long range forces (`md`)
- 9. *Generic direct N-body solvers* with long range forces (for vortices) (`n-body`)
- 10. *Particle-in-cell* in two dimensions:
 - (a) straightforward implementation (`pic-simple`)
 - (b) sophisticated implementation (`pic-gather-scatter`)
- 11. *QCD kernel*: staggered fermion Conjugate Gradient method (`qcd-kernel`)
- 12. *Quantum Monte-Carlo* (`qmc`)
- 13. *Quadratic programming* (`qptransport`)
- 14. *Solution of nonsymmetric linear equations* using the Conjugate Gradient Method (`rp`)
- 15. *Euler fluid flow in two dimensions using an explicit finite difference scheme* (`step4`)
- 16. *Wave propagation in one dimension* (`wave-1D`)

This document provides an overview of the DPF benchmark suite. The details required for the use of the suite are covered online at <http://www.das.harvard.edu/cs/research/dpf/root.html>. Sources, examples of usage, and the produced output of the DPF benchmarks are also available there.

The intent of this documentation is to provide sufficient insight into the benchmark codes for prospective users to choose one or a subset of codes that would best expose and measure specific features of a compiler or system. We describe the employed data structures and their layout, the floating-point operation count, the dominating communication patterns, the primary local memory access patterns, and some characteristics of the implementation.

1.2 Architectural Model

Constructing a benchmark suite suitable for compiler evaluation requires both a hardware model and an execution model in addition to a language definition. Most benchmarks in the DPF suite are appropriate for any parallel architecture, whether the memory is distributed among the processors or shared and whether the underlying programming model is distributed or shared memory.

Some of the benchmarks focus on evaluating how well the local memory hierarchy (consisting of registers and caches and main memory) is utilized in a distributed memory multiprocessor. For example, in the Connection Machine CM-5 [63], the nodes are best viewed as having a register-oriented vector architecture with respect to the data-parallel programming model. However,

the main memory consists of dynamic random access memory (DRAM) that is operated in page mode. Thus, in order not to incur page faults, blocking of nested loops is often necessary. Moreover, main memory is addressed through a translation table that holds a strict subset of DRAM pages. Optimum performance for some computations therefore requires yet another level of blocking. Several of the Connection Machine Scientific Software Library (CMSL) [65] routines (used in some of the DPF codes) attempts to exploit locality of reference in this four level hierarchy (registers, DRAM pages, mapped pages and remote memory). Hence, for other platforms with similarly optimized library codes, executing a DPF benchmark in both a non-library and a library version would gain insight into the effectiveness of the relevant compiler optimizations. Other codes in the DPF suite focus on remote data references and will expose properties of communications libraries and corresponding run-time system features.

We also expect the DPF benchmarks to be useful for the evaluation of compilers and supporting software systems on the so-called distributed shared memory architectures, such as the Stanford Dash and Flash architectures [44], the SGI Origin 2000, and the HP/Convex SPP/2000. For such memory architectures, the parallel code efficiency is affected by how the data reference pattern interplays with maintaining cache coherence through hardware [43, 44], run-time system [57], or compiler [56] features.

Yet another group of the DPF codes contain constructs related to an execution model in which one processor is primarily responsible for the execution control of single-thread programs, such as a typical HPF (no extrinsic procedures) and CMF (no local-global features) program. Such processors act very much in the same fashion as the scalar unit in a typical vector processor.

1.3 Language Aspects

Our target language, HPF, is an extension of Fortran-90. The main differences are a set of layout statements for explicit management of data distribution and the *FORALL* construct. The DPF benchmark suite contains the *FORALL* construct in a variety of situations: applied to data within individual processors only (that is, no communication required), applied to a mixture of local and remote data, with and without masks, etc. All codes using arrays of two or more dimensions include layout directives, which affect the storage-to-sequence association.

Some of the new constructs in Fortran-90 (compared to Fortran-77) specify data references on whole arrays, or segments thereof, such as *CSHIFT*, *EOSHIFT*, *SPREAD* and *SUM*. In the context of distributed memory architectures, these functions define *collective communication* patterns, and are often implemented as library functions, whether explicitly available to the user (as part of the run-time system supporting the compiler) or only callable by the compiler. By specifying the collective communication explicitly, path selection and scheduling can be optimized without sophisticated code analysis. Another Fortran-90 construct is the triplet notation for array sectioning in expressions, which, in general, imply both local memory data motion and communication. The Fortran-90 and HPF unique constructs used in the DPF benchmark suite are summarized in Table 1.

Fortran-90 memory management is more complex than in Fortran-77, and its effective handling is critical for any compiler and associated run-time system for high performance systems. Thus, Fortran-90 offers both heap and stack based dynamic array allocation, and allows the programmer to declare arrays as static, allocatable, or automatic. Dummy array arguments passed in subroutine calls allow for run-time memory management in the form of adjustable, assumed-shape and assumed-size arrays. The DPF codes all declare explicit-shape

Language Constructs	Array Data Structures			
	1-D	2-D	3-D	4-D,5-D,6-D,7-D
Fortran-90 constructs				
CSHIFT	pcr(1) conj-grad jacobi fft-1D qptransport wave-1D	pcr(2) jacobi fft-2D ellip-2D n-body step4	pcr(3) fft-3D boson rp	mdcell qcd-kernel
EOSHIFT	qptransport			
TRANPOSE		transpose		
Array sections	diff-1D	diff-2D	diff-3D	
SPREAD		matrix-vector(1) qr-factor/qr-solve gauss-jordan jacobi md n-body	matrix-vector(2,3,4)	qmc
SUM		matrix-vector(1) qr-factor/qr-solve ellip-2d ks-spectral md qmc	matrix-vector(2,3,4) mdcell rp	
MAXVAL			mdcell	
MINVAL			mdcell	
MAXLOC		gauss-jordan		
WHERE	fft-1D	qr-factor/qr-solve jacobi fft-2D	fft-3D fem-3D mdcell pic-gather-scatter	
Mask in SUM		qr-factor/qr-solve		
Mask in MAXLOC		gauss-jordan		
HPF constructs not in Fortran-90				
FORALL		gather scatter transpose jacobi diff-2D gmo pic-simple	mdcell pic-gather-scatter	
Mask in FORALL	boson	fermion pic-simple	pic-gather-scather	
Prefix	qptransport	qmc	pic-gather-scather	
Layout directives	all codes			
CMF utility functions with correspondence in HPF intrinsics				
CMF_send_add CMF_send_overwrite		qmc	pic-gather-scather	
CMF_aset_ld			mdcell	
CMF_order	pic-gather-scather			

Table 1: Summary of Fortran-90 and HPF language constructs in the DPF benchmark codes

arrays that either are static or automatic, and the dummy arrays are all adjustable, except in a few cases where static arrays are declared.

Another aspect of memory management is striding through local memory as defined by loop nests and their index strides. Fortran-90 supports four modes of local memory access: implicit (referring to the entire axis range), direct (axis is indexed directly by the loop variable), indirect (axis is indexed by another array) and strided (axis is indexed by a triplet subscript). We describe the array declarations and array element access modes for the various codes.

In some execution models, like the one used for CMF and all vector architectures, one processor, the Control Processor, (for vector architectures the scalar processor) performs scalar operations and instruction storage and broadcast. Therefore, the use of scalar variables in array expressions often implies communication between the Control Processor and the processing elements. Since this communication is critical for all operations, the communication network between the Control Processor and the processing elements is often of a higher capacity than the network between the processing elements, but not of as high a capacity as that between individual processing elements and their associated memory. Several of the DPF codes contain constructs that imply communication between the Control Processor and the processing elements. The DPF **n-body** code contains constructs that allow for a clear comparison between the two networks: Control Processor to and from processing elements, and between processing elements.

Since the DPF benchmark suite was developed before HPF compilers were available, it was written in CMF, for which a fairly mature compiler existed at the time of this project. In addition, the CM-5 platform had a library designed for data parallel languages and well optimized for scalable architectures. This permitted verification and testing of the benchmark suite in the intended setting. All of the CMF utility functions used in the DPF benchmarks, however, have their correspondence in HPF intrinsics, as shown in Table 1.

In view of the fact that the intended target language for the DPF suite is HPF, we adhere to the HPF terminology. For instance, the CMF layout directives, **serial** and **new**, although retained in the code segments, are referred to as *local* and *parallel* throughout this document.

The performance evaluation and analysis is based on the execution semantics of HPF. Thus, the execution of the statement **vtv = sum(v*v, mask)**, implies that the self inner-product of the vector **v** is executed for all elements, rather than only the unmasked ones. As for the summation that is performed under mask, we only count operations associated with mask elements being *true* when the mask is data independent (predictable at compile time). Otherwise, we count operations as if all elements of the mask were true.

1.4 DPF Code Versions

A number of the benchmarks exist in several forms. A **basic CMF** version is provided in most cases. This version is intended to represent code produced by a knowledgeable user without a lengthy optimization process, that is, a “typical” user code. In some cases an **optimized CMF** version is also provided, representing the kind of code a highly performance oriented, advanced programmer with good knowledge about the compiler and the architecture would produce.

Optimization can also be achieved via calls to highly optimized routines, which may be implemented in languages other than CMF. Such routines are typically found in a run-time system library, a scientific software library or, in the case of the Connection Machine systems, in the CMF library augmenting the intrinsic functions of the language. For optimizations

resorting to CMF library functions, the code version is termed **library CMF**, whereas for those codes calling the CMSSL functions, we refer to the code version as **CMSSL**. In other cases, rather than resorting to library calls, some segment of the code, critical to the benchmark performance, is identified and implemented in the lower level language C/DPEAC [66]. This code version is termed **C/DPEAC** and is assumed to give the programmer finer control over the underlying architecture. Some DPF benchmarks provide several versions at the same level of optimization. In such cases, we refer to the code version by indexing it, such as **transpose(1)** and **transpose(2)** in Table 5. A summary of the available code versions is provided in Table 2.

Benchmark Name	basic CMF	opt. CMF	lib. CMF	CMSSL	C/DPEAC
gather	×				
scatter	×				
reduction	×				
transpose	×			×	
matrix-vector	×			×	
lu				×	
qr	×			×	
gauss-jordan	×				
pcr	×			×	
conj-grad	×				
jacobi	×				
fft	×			×	
boson	×				
diff-1D				×	
diff-2D				×	
diff-3D	×				
ellip-2D	×				
fem-3D				×	
fermion	×				×
gmo		×			
ks-spectral	×			×	
mdcell	×				
md	×				
n-body	×				
pic-simple				×	
pic-gather-scatter			×		
qcd-kernel		×			
qmc			×		
qptransport			×		
rp	×				
step4	×				
wave-1D	×			×	

Table 2: The code versions in the overall benchmark suite

The online documentation gives information about how to run the programs and for each code the meaning of the arguments, memory requirements as derived from array declarations and layout directives (excluding temporary arrays generated by the compiler and the run-time system), and floating-point operations as a function of input arguments in order to allow for an estimate of the resources and time required to run the code. In all, there are 32 benchmarks in the suite, comprising about 17,000 lines of source code. The full DPF benchmark suite (including the sample data files) occupies 2.64 MBytes.

Operation Type	FLOPs
$+$, $-$, $\times$	1
$\div$, $\sqrt{\quad}$	4
$\log$, $\exp$, $\sin$, $\cos$,...	8

Table 3: Number of FLOPs for each operation type

1.5 Performance Evaluation

The majority of the benchmark codes report the following performance metrics:

CM-busy time (in seconds): non-idle time spent in executing the benchmark.

CM-elapsed time (in seconds): total time (idle and non-idle) spent in executing the benchmark.

Busy floprate (in MFLOPs): number of million floating-point operations per second obtained by dividing the stated FLOP count for the benchmark by the CM-busy time.

Elapsed floprate (in MFLOPs): number of million floating-point operations per second obtained by dividing the stated FLOP count for the benchmark by the CM-elapsed time.

In some of the application codes, namely `boson`, `fem-3D`, `md`, `mdcell`, `pic-gather-scatter`, `qcd-kernel`, `qptransport` and `step4`, the above measures are provided for code segments, rather than the entire benchmark. Similarly, performance metrics for different modules of a benchmark may also be reported separately. For instance, the factorization and solution times for `qr` and `lu` are reported separately. Similarly, in `diff-1D` and `diff-2D`, the constituents of the dominant loop are timed separately.

Where applicable, we attempt to quantify performance using the following attributes:

FLOP count: In counting the number of FLOPs we adopt the operation counts suggested in [22], and summarized in Table 3, for addition, subtraction, multiplication, division, square root, logarithms, exponentials and trigonometric functions. For reduction and parallel prefix operations, such as the intrinsic *SUM* and segmented *Scans*, we use the sequential FLOP count, i.e., $N - 1$ for N element 1-D arrays.

Arithmetic Efficiency (in percent): The arithmetic efficiency is computed by dividing the busy FLOP rate by the peak FLOP rate of all the participating processors². This performance measure is only computed for the linear algebra functions.

Memory usage (in bytes): The reported memory usage only covers user declared data structures including all the auxiliary arrays required by the algorithm's implementation, given the data type sizes of Table 4. Temporary variables and arrays that may be generated by the compiler are not accounted for.

In the case where a lower dimensional array L is aligned with a higher dimensional array H , L effectively takes up the storage of $size\{H\}$ and we report the collective memory of L and H to be $2\ size\{H\}$.

²In the case of the CM-5, the peak FLOP rate is 32 MFLOPs per second per vector unit (VU) and for the CM-5E it is 40 MFLOPs per second.

Data Type	Size	Symbol
integer	4	t
logical	4	l
single-precision real	4	s
double-precision real	8	d
single-precision complex	8	c
double-precision complex	16	z

Table 4: Data type sizes (in bytes) and symbols for standard 32-bit based arithmetic architectures

Communication pattern: We specify the types of communication an algorithm exhibits, as well as the language constructs with which they are expressed. These communication types include stencils, gather, scatter, reduction, broadcast (*SPREAD*), all-to-all broadcast (AAB) [39], all-to-all personalized communication (AAPC) [39], butterfly, *Scan*, circular shift (*CSHIFT*), send, get, and sort. It should be noted that more complex patterns (such as stencils and AAB) can be implemented by more than one simpler communication function (for instance *CSHIFTs*, *SPREADs*, etc.).

Operation count per iteration (in FLOPs): We give the number of floating-point operations per data point obtained by dividing the total FLOP count of the benchmark by the problem size. This metric serves as the first approximation to the computational grain size of the benchmark, giving an insight into how the program scales with increasing problem sizes.

Communication count per loop iteration: We group the communication patterns invoked by this benchmark and specify exactly how many such patterns are used within the main computational loop. This metric, together with the operation count per iteration, give the relative ratio between computation and communication in the benchmark.

1.6 Benchmark Guide

The salient features of the DPF benchmark codes are summarized in a set of tables as follows.

Language constructs: Table 1.

Data Layout: Table 6 for library functions and Table 15 for application kernels.

Local memory accesses, types of array allocation: Table 5 for communication codes, Table 9 for linear algebra codes, and Table 19 for application codes.

Collective communications: The use of certain types of collective communications is covered by Tables 7 and 16 for linear algebra and application kernels, respectively. Implementation techniques for some of the communications in the application kernels are covered in Table 17.

Computation/communication ratio, memory usage: The number of arithmetic operations per communication as well as the benchmark’s memory usage is listed in Table 8 for library codes and in Table 18 for application kernels.

Since individual processor performance is critical for high performance on any scalable architecture, the DPF benchmark suite includes three codes that do not invoke any interprocessor communication: **matrix-vector(3)**, **fermion** and **gmo**.

Some of the language features for which one might like to evaluate compiler optimizations appear both in “isolation”, as well as in some application context in the DPF benchmark suite. Separate timing of the constructs within application kernels can be easily obtained by inserting timing statements as desired, since the source code is available. For instance, gather and scatter operations appear in separate benchmarks, **gather** and **scatter**, as well as in application kernels such as **fem-3D** and **pic-gather-scatter**. AAPC appear in the form of matrix transposition in **transpose** and **diff-2D**, and for certain array sizes also in the form of bit-reversal in **fft**. The **gather** and **scatter** benchmarks allow for the investigation of arbitrary communication patterns, including replication, and some control of the relative locations of source and destination array locations of individual elements.

Broadcast, reduction and nearest neighbor array communication are important programming primitives that appear in several benchmarks, some of which largely depend on these primitives for efficient execution. For instance, on most architectures, the matrix-vector multiplication benchmark will be dominated by the time for broadcast (*SPREAD*) and reduction (*SUM*). *SPREAD* is also likely to dominate the execution time for the Gauss-Jordan matrix inversion benchmark, while *CSHIFT* is expected to have a significant impact on the performance for both tridiagonal system solvers, the FFT, and several of the application kernels implementing stencils through *CSHIFTs*, such as **boson**, **ellip-2D**, **mdcell**, **rp** and **wave-1D**.

The linear algebra library subset of the DPF benchmark suite is provided to enable testing the performance of compiler generated code against that of a highly optimized library, such as the CMSSL for the CM-5. Performance attributes for the linear algebra codes are presented in Tables 6, 7 and 8, tabulating the data representation and layout for dominating kernel computations, the communication pattern along with their associated array ranks, and the computation to communication ratio in the main loop. These tables can be used to decide on an appropriate benchmark code according to a given testing criteria. For instance, if a user desires to evaluate how a particular compiler implements *CSHIFT* on a 2-D array, Table 8 indicates that there are three choices: **jacobi**, **fft 2-D** and **pcr(2)**. If having local axis (and therefore local memory addressing) in the main computational kernel is desired (or not), then by Tables 6 and 8 the **pcr(2)** is picked (or ruled out). In the latter case, to decide between the **jacobi** and **fft-2D** codes, Table 8 shows the other communication patterns associated with each. Hence, if minimizing other communication patterns is the goal, the **fft-2D** code would be selected, especially since the performance of the AAPC can be measured via the **transpose** code. Conversely, if evaluating broadcasts and sends is also of interest, then **jacobi** would be the appropriate choice.

The application kernels of the benchmark suite is intended to cover a wide variety of scientific applications typically implemented on parallel machines. Table 14 captures some of the essential features by which many application codes are classified. Many finite difference codes using explicit solvers require emulation of grids on one or several dimensions. Depending on the layout of the associated arrays, the interprocessor may or may not correspond directly to the dimensionality of the data arrays. Table 14 specifies the interprocessor communication implied in the application kernels. The application kernels contain one code for unstructured grid computations, **fem-3D**, pure particle codes, **n-body** and **md**, and codes that make use of both regular grid structures and particle representations, **mdcell**, **pic-simple** and **pic-gather-scatter**.

Tables 14, 15, 16, 17, 18 and 19 can be used to aid in the selection of one code or a subset of benchmark codes for a specific task. For instance, for indirect addressing we have four choices according to Table 19: `gather`, `scatter`, `mdcell` and `pic-gather-scatter`. The same table also shows that `gather` and `scatter` perform indirect addressing on a parallel axis, while `mdcell` and `pic-gather-scatter` perform indirect addressing on a local axis. If, for example, an application code with an AAB is desired, then Table 16 yields the codes `md` and `n-body`. Table 15 shows that only `n-body` is guaranteed to perform an AABC with respect to the processing elements, since one of the axes is local while both are distributed in `md`. Both codes perform AAB communication with respect to the index space. On the other hand, Table 17 states that if the implementation of AAB in the form of *SPREADs* is of interest, then `md` is the code of choice. Alternatively, `n-body`'s AAB includes *CSHIFTs* and a *Broadcast* from the Control Processor.

2 Library Functions for Communication

The communication library functions, except the *reduction* function, do not perform any floating-point operations. Hence, with exception of reduction, the FLOP count is not applicable to this set of benchmarks and it is not reported for any code in the set. The communication pattern is, on the other hand, described more fully and the memory usage is also given in each case. A summary of the local memory access mechanisms and array types for the communication codes is given in Table 5.

Code	Local Memory Access		Array Allocation Types	
	:serial	:news	Declaration	Dummy Array
<code>gather</code>	1 axis: implicit	indirect	automatic	adjustable
<code>scatter</code>	1 axis: implicit	indirect	automatic	adjustable
<code>reduction</code>	N/A	implicit	automatic	N/A
<code>transpose(1)</code>	N/A	implicit	automatic	N/A
<code>transpose(2)</code>	1 axis: implicit	implicit	automatic	N/A

Table 5: Local memory access mechanisms and types of array allocation for the communication codes

The `gather` and `scatter` benchmark codes are designed to allow for the evaluation of blocking and pipelining of communications, caching of address computations as well as efficiency in replication. The benchmarks also attempt to measure the significance of the “complexity” of the communication pattern by providing some control over the relative locations of sources and destinations as well as the total number of elements referenced. Both the source and destination arrays are 2-D with one local axis. `Gather` and `scatter` operations are only performed on the parallel axis. All elements of the serial axis are subject to the same `gather/scatter` operation. Thus, except for the local memory offset, it suffices to perform one address computation, and potentially blocks of size equal to the local memory extent could be communicated. Furthermore, since the extent of the parallel axis can exceed the number of processing nodes, the block size could be larger, although there is no guarantee that parallel axis elements assigned to the same processor have sources in the same processor.

2.1 Gather

Given a source array of shape $b \times s$, a destination array of shape $b \times d$, and an index vector of extent d , a *gather* operation is performed on the axis of extent s into the axis of extent d of the destination array, as specified by the index vector. The source and destination arrays are laid out with the axis of extent b local to a processing node. Thus, the axis of size b allows for multi-gather. The gather operation is expressed as follows:

```
forall (k=1:b, j=1:d) dest(k, j) = source(k, index(j))
```

For single-precision real arrays, the user allocated memory for the gather function is $4b(s + d) + 4d$ bytes.

The benchmark function allows five different communication modes to be specified.

1. **Cyclic:** The cyclic mode is defined by $dest(:, j) \leftarrow source(:, 1 + (j - 1) \bmod s)$. Thus, for $d > s$ elements of the source array are replicated into the destination array with a stride of s , i.e., the source array is replicated cyclicly with the length of the cycle being the size of the source array. Note that if $d \leq s$, then the cyclic mode degenerates to the identity mapping of a segment of the source array to the destination array, starting at the beginning of the parallel axis of both arrays.
2. **Block:** If $d > s$, elements of the source array are replicated into blocks of length d/s in the destination array. The replication is performed as follows: $source(:, i)$ is sent to all $dest(:, j)$ such that $(d/s) \cdot (i - 0.5) \leq j < (d/s) \cdot (i + 0.5)$. If $d \leq s$, the cyclic and block modes are identical.
3. **Random:** For each destination array index, a source array index is selected randomly using a uniform distribution. Note that replications may occur, even when $d \leq s$.
4. **Collision:** This mode is only valid when $d \geq s$. First, a subset of $s/\text{contention}$ different source array indices of the parallel axis of extent s are selected randomly, where *contention* is an integer input parameter. Only these indices participate in gathering data for the destination array. Then, for *each* destination array index, one of the previously selected $s/\text{contention}$ source array indices is selected randomly, so replication occur. Specifying smaller values of s relative to d achieves the same effect in increasing replication as increasing the *contention* parameter. However, the two options have a different effect on network contention since for the same replication the relative locations of sources and destinations are different in the two alternatives of specifying the set of source indices. When *contention* = 1, this mode degenerates to the random mode.
5. **Semi-local:** This mode is only valid for $d = s$. The mapping is defined by $dest(:, j) \leftarrow source(:, 1 + (j + \text{offset} + s - 1) \bmod s)$, where *offset* is selected randomly from the interval $[-L, L]$. The size $L = d(1 - \text{locality})$ of the locality window is determined by the real input parameter *locality*, which must lie in the interval $[0, 1]$. When *locality* = 0, this mode degenerates to the random mode; when *locality* = 1, it degenerates to an identity mapping.

Communication primitives similar to this **gather** benchmark appear frequently in basic linear algebra operations for arbitrary sparse matrices, for histogramming and many other applications, such as finite element codes for unstructured grids.

2.2 Scatter

Given a source array of shape $b \times s$, a destination array of shape $b \times d$, and an index vector of extent s , a *scatter* operation is performed on the axis of extent s into the axis of extent d of the destination array, as specified by the index vector. $d \geq s$ must hold for this benchmark. The source and destination arrays are laid out with the axis of extent b local to a processing node. The scatter operation is expressed as follows:

```
forall (k=1:b, i=1:s) dest(k, index(i)) = source(k, i)
```

If a conflict-free scatter is desired, then the index vector must have unique values, i.e., the same value cannot appear in more than one element of the index vector.

For single-precision real arrays, the user allocated memory for the scatter function is $4b(s + d) + 4s$ bytes.

The benchmark function allows four different communication modes to be specified. All but the last mode are conflict-free.

1. **Identity:** An identity mapping of the source array to the destination array is performed: $source(i) \rightarrow dest(i)$. When $d > s$, the superfluous segment of the destination array does not participate in the operation.
2. **Stride:** The source array is copied into the destination array with a stride equal to the relative sizes of the arrays parallel axes: $source(i) \rightarrow dest(i \cdot \lfloor d/s \rfloor)$.
3. **Random:** For each source array index, a destination array index is selected randomly using a uniform distribution. The random selection is performed in a conflict-free manner.
4. **Collision:** First $s/\text{contention}$ different destination indices of the parallel axis of extent d are selected randomly, where *contention* is an integer input parameter (with default 10). Only these indices of the destination array receive data. Then, for *each* source array index, one of the previously chosen $s/\text{contention}$ destination indices is selected randomly for each source array index, so conflicts arise. Because this mode is not conflict-free, it is implemented with a *FORALL* statement using the intrinsic *SUM* for conflict resolution through reduction. The summation is masked to include only those source locations that are mapped to the particular destination location. An outer sequential do-loop iterates over the axis of extent b , i.e., the axis local to the processors.

Communication primitives similar to this **scatter** benchmark appear frequently in basic linear algebra operations for arbitrary sparse matrices, for histogramming and many other applications, such as finite element codes for unstructured grids.

2.3 Reduction

The functionality of the **reduction** benchmark corresponds to the intrinsic function *SUM*. The reduction operation is performed on 1-D and 2-D square arrays of single-precision real data type. It is expressed with one line of code:

```
total = sum(A)
```

where **A** is a 1-D or 2-D array and **total** is a real variable in which the sum of all the elements of **A** is stored. Arrays of sizes up to 64 Mbytes are tested. By varying the size of *A* relative to the number of nodes, the efficiency of local and global summations can be assessed.

Global reduction operations are common in many library functions and applications. This benchmark function allows for the evaluation of the efficiency of such global operations which are included as intrinsic functions in some languages.

2.4 Transpose

The **transpose** benchmark carries out a transposition of a square matrix laid out with one axis local to the processors and one axis parallel, i.e., distributed across processors. There are two basic versions of the benchmark code:

1. Transposition through the use of the *FORALL* statement.

```
forall (i = 1:n, j = 1:n) x(i,j) = x(j,i)
```

2. or transposition through the use of the Fortran-90 intrinsic *TRANSPOSE*.

```
x = transpose(x)
```

where **x** is an $n \times n$ array laid out as

```
CMF$ layout x(:serial,)
```

The DPF benchmark also contains a library version of this benchmark. This version calls the CMSSL for the transposition.

For an $n \times n$ array, the user allocated memory is cn^2 bytes, where c is 4, 8 or 16, depending on whether the data type of the array is single-precision real, single-precision complex, or double-precision complex, respectively. However, n is not specified by the user but determined at run-time based on the amount of available memory. Its maximum value is $n = \sqrt{fM/c}$, where fM is some fraction of the total available memory M (f is defined in the benchmark code). Although the array transposed is a fraction of the memory, additional space is needed for temporaries created by the compiler, therefore all the available memory may be consumed.

The communication pattern portrayed is global-local transpose, which is essentially all-to-all personalized communication (AAPC). Thus, apart from being an indispensable operation in linear algebra and other numerous applications, the benchmark may be used to measure the bisection bandwidth, and was studied in a class of permutations in [15, 60].

3 Library Functions for Linear Algebra

The Connection Machine Scientific Software Library (CMSSL) was created for data parallel languages and distributed memory architectures. CMSSL attempts to make efficient use of the underlying system architecture through efficient implementation of interprocessor data motion and optimal management of local memory hierarchy and data paths in each processor. Since these are precisely the issues investigated in modern compiler design for parallel languages and

Code	Arrays			
	1-Dimensional	2-Dimensional	3-Dimensional	4-Dimensional
matrix-vector: (1)	X(:)	X(:, :)	X(:, :, :)	
(2)		X(:, :)	X(:, :, :)	
(3)		X(:serial, :)	X(:serial, :serial, :)	
(4)		X(:, :)	X(:serial, :, :)	
lu			X(:, :, :)	
qr		X(:, :)		
gauss-jordan	X(:)	X(:, :)		
pcr: (1)	X(:)	X(:serial, :)		
(2)		X(:, :)	X(:serial, :, :)	
(3)			X(:, :, :)	X(:serial, :, :, :)
conj-grad	X(:)			
jacobi	X(:)	X(:, :)		
fft: 1-D	X(:)			
2-D		X(:, :)		
3-D			X(:, :, :)	

Table 6: Data representation and layout for dominating computations in linear algebra kernels (“:serial” represents a local axis, “:” represents a parallel axis)

on parallel machine architectures, the library subset of the DPF benchmark suite is provided to enable testing the performance of compiler generated code against that of any highly optimized library, such as the CMSSL.

The subset included in the DPF suite is comprised of matrix-vector multiplication, dense matrix solvers, two different tridiagonal system solvers (based on parallel cyclic reduction and the conjugate gradient method respectively), a dense eigenanalysis routine and an FFT routine. Where possible, the interface conventions are kept identical with those of CMSSL. In many cases, different layouts are accepted and analyzed before calling the common interface. Documentation of the CMSSL interface with all its conventions can be found in [65].

The description of each benchmark function includes a FLOP count, the amount of user allocated memory, and a characterization of data reference patterns for remote and local references.

We summarize some of the important properties of our implementations of the linear algebra benchmarks by means of three tables. Table 6 gives an overview of the data representation and layout for the dominating computations. Table 7 shows the communication operations used along with their associated array ranks. Table 8 tabulates the computation to communication ratio in the main loop of each linear algebra benchmark. The access pattern for the local axes of the arrays in the main loop of the benchmark and the array allocation mechanisms are summarized in Table 9.

3.1 Matrix-Vector Multiplication

This DPF benchmark is a collection of routines computing one or more matrix-vector products. Given data parallel arrays $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{A}$ containing *multiple instances* [38] of the vectors x and y and the matrix A , respectively, the matrix-vector routines perform the operation $y \leftarrow y + Ax$ for each instance. The benchmark kernels support all four standard floating-point data types.

The following code fragment shows the implementation of the DPF benchmark.

Communication Pattern	Arrays		
	1-Dimensional	2-Dimensional	3-Dimensional
CSHIFT	pcr (1) conj-grad jacobi fft-1D	pcr (2) jacobi fft-2D	pcr (3) fft-3D
SPREAD		matrix-vector (1) qr-factor/qr-solve gauss-jordan jacobi	matrix-vector(2,3,4)
SUM		matrix-vector (1) qr-factor/qr-solve	matrix-vector(2,3,4)
MAXLOC		gauss-jordan	
Send/Get		gauss-jordan jacobi	

Table 7: Communication pattern of linear algebra kernels (the array dimensions for reduction and broadcast are of source and destination respectively)

Code	FLOP Count (per iteration)	Memory Usage (in bytes)	Communication (per iteration)
matrix-vector	s,d: $2nmi$ c,z: $8nmi$	s: $4(n + nm + m)i$ d: $8(n + nm + m)i$ c: $8(n + nm + m)i$ z: $16(n + nm + m)i$	1 Broadcast, 1 Reduction
lu: factor lu: solve	$2/3 n^2 i$ $2rni$	d: $8n(n + 2r)i$ d: $8n(n + 2r)i$	1 Reduction, 1 Broadcast 1 Reduction
qr: factor	s,d: $(5.5m - 0.5n)n$ c,z: $4(5.5m - 0.5n)n^2$	s: $24mn$ d: $36mn$ c: $40mn$ z: $68mn$	2 Reductions, 2 Broadcasts
qr: solve	s,d: $(8m - 1.5n)n$ c,z: $4(8m - 1.5n)n^2$	s: $24mn + 4m(r + 1)$ d: $44mn + 8m(r + 1)$ c: $48mn + 8m(r + 1)$ z: $92mn + 16m(r + 1)$	2 Reductions, 4 Broadcasts
gauss-jordan	$n + 2 + 2n^2$	s: $28n^2 + 16n$	1 Reduction, 3 Sends, 2 Gets, 2 Broadcasts
pcr	s,d: $(5r + 12)ni$ c,z: $4(5r + 12)ni$	s: $4(r + 4)ni$ d: $8(r + 4)ni$ c: $8(r + 4)ni$ z: $16(r + 4)ni$	$(2r + 4)$ CSHIFTS
conj-grad	$15n$	d: $40n$	4 CSHIFTS, 3 Reductions
jacobi	$6n^2 + 26n$	s: $44n^2 + 28n$ d: $88n^2 + 4n$	2 CSHIFTS on 1-D arrays, 2 CSHIFTS on 2-D arrays, 2 Sends, 4 1-D to 2-D Broadcasts
fft-1D	$5n$	c: $60n$ z: $100n$	2 CSHIFTS
fft-2D	$10n^2$	c: $76n^2$ z: $115n^2$	4 CSHIFTS
fft-3D	$15n^3$	c: $92n^3$ z: $136n^3$	6 CSHIFTS

Table 8: Computation to communication ratio and memory usage in linear algebra kernels

Code	Local Memory Access		Array Allocation Types	
	:serial	:news	Declaration	Dummy Array
matrix-vector(1,2,4)	1 axis: implicit	implicit	automatic	adjustable
matrix-vector(3)	2 axes: implicit	implicit	automatic	adjustable
qr: factor	N/A	direct, implicit	automatic	adjustable
qr: solve	N/A	direct, implicit	automatic	adjustable
gauss-jordan	N/A	implicit	automatic	adjustable
pcr	1 axis: direct	implicit	automatic	adjustable
conj-grad	N/A	implicit	automatic	adjustable
jacobi	N/A	implicit	automatic	adjustable
fft-1D	N/A	implicit	automatic	adjustable
fft-2D	N/A	implicit	automatic	adjustable
fft-3D	N/A	implicit	automatic	adjustable

Table 9: Local memory access mechanisms and types of array allocation for the linear algebra codes

```

real*4 y(m,1), A(m,n,1), x(n,1)
real*4 xx(m,n,1)

xx = A * spread(x,1,m)
y = y + sum(xx,2)

```

The matrix-vector multiplication is implemented for the following layouts:

1. one instance of the three operands with each instance spread over all the processors,
2. multiple instances with each instance of each operand occupying a subset of the processors,
3. multiple instances with each instance of the corresponding operands allocated to the memory unit associated with one processor,
4. multiple instances with the row axis (the axis crossing different rows) of array **A** allocated local to the processors, and the other axis of **A** as well as the axes of the other operands spread across the processors.

The third layout requires no communication and represents a truly embarrassingly parallel case, while the fourth case only requires communication during the reduction. All four matrix-vector multiplication routines use the same algorithm and implementation thereof. Only the layout differs.

With each matrix A of shape $p \times q$ and i instances, the FLOP count is $2pqi$ for real data types and $8pqi$ for complex data types. The user allocated memory in bytes is $4(p + pq + q)i$, $8(p + pq + q)i$, $8(p + pq + q)i$, or $16(p + pq + q)i$ for single-precision real, double-precision real, single-precision complex and double-precision complex, respectively.

The execution time for this implementation of the matrix-vector multiplication will for most architectures be dominated by the *SPREAD* and *SUM* operations, and the implicit alignment of the input and output array x with A and the result array `sum(xx,2)` with y .

Matrix-vector multiplication is a typical level-2 BLAS operation. It is the dominating operation in iterative methods for the solution of linear systems of equations. It only requires two floating-point operations per matrix element and its performance is very sensitive to data

motion. In case one above, each operand is distributed across all nodes such that the input vector must be aligned with the matrix, and the result vector aligned with the output vector as part of the computation.

3.2 LU Factorization and Solution

This benchmark is only available in the CMSSL form. Given i instances of a dense square matrix A of size $n \times n$, and, for each instance, r right hand side vectors of size n in the form of an $n \times r$ matrix B , these routines solve the i instances of the dense system of equations $AX = B$ by factoring each instance of the matrix A into a lower triangular matrix L and an upper triangular matrix U , such that $A = LU$. The factorization method is Gaussian elimination with or without partial pivoting. Each factored system is solved using forward elimination followed by back substitution.

The details of the parallel implementation are given in [45, 65]. *SPREADs* are used for the rank-one or higher updates, and reduction is used for the selection of the pivot row. Whereas load-balance is an issue for all phases of the algorithm, the “ $O(n^3)$ ” operations can be load-balanced fairly easily, while the “ $O(n^2)$ ” operations do not load-balance well [45]. Load-balance for the block allocation used by the CMF compiler is accomplished through a block-cyclic elimination order.

The `lu` benchmark uses 3-D double-precision real arrays. Array A is of shape $n \times n \times i$ and array B is of shape $n \times r \times i$. Moreover, an auxiliary array is used for the creation of the initial B , also of shape $n \times r \times i$. Thus, the user allocated memory for this benchmark is $8n(n + 2r)i$ bytes. The factorization and solution phases approximately consume $(2/3)n^3i$ FLOPs and $2rn^2i$ FLOPs respectively, bringing the overall FLOP count of the benchmark to a total of $((2/3)n + 2r)n^2i$ FLOPs, approximately³.

The single instance version of this benchmark corresponds to the well known LINPACK [14] benchmark.

3.3 QR Factorization and Solution

This benchmark is a collection of routines for solving dense linear systems of equations using Householder transformations [12, 20], commonly referred to as the *QR routines*. Given an $m \times n$ coefficient matrix A , where $m \geq n$, and a set of r right hand side vectors in the form of an $m \times r$ matrix B , the QR routines factorize and solve the system of equations $AX = B$. The matrix A is factored into a trapezoidal matrix Q and a triangular matrix R , such that $A = QR$. Then, the solver uses the factors Q and R to calculate the least squares solution to the system $AX = B$, i.e., to compute the set of r vectors in array X (each corresponding to a particular right hand side).

The CMF version of the QR routines only supports single-instance computation and performs the Householder transformations *without* column pivoting. The corresponding CMSSL version implements a more general interface. Below, we discuss the CMF version. For a detailed description of the CMSSL version see [65].

Both the factorization and the solution routines make use of masks. An alternative would be to use array sections. Whichever approach yields the highest performance and requires the least

³The FLOP counts include higher order terms only.

memory depends upon how the compiler handles masked operations, the penalty for carrying out operations under masks, and how array sections are implemented, in particular with respect to temporary storage. For instance, masked operations are often carried out everywhere, but only results where the mask is true are stored. The alternative is to carry out the operation only where the mask is true. The latter is the underlying assumption in counting the number of FLOPs (Section 1.5).

The communication patterns appearing are *SPREADs* and reductions. The reductions are performed within the intrinsic function *SUM*. In the solution routines, the right hand side matrix B is aligned with A through assignment to another array (`rhs`) of the same shape as A . Therefore, the misalignment overhead is reduced, at the expense of additional memory space. Due to this alignment, $n \geq r$ must hold. Thus, for the CMF implementation of the DPF QR benchmark, the matrix sizes must satisfy the inequality $m \geq n \geq r$.

FLOP Count	Factorization	Solution	Total
Real	$(5.5m - 0.5n)n^2$	$(8m - 1.5n)n^2$	$(13.5m - 2n)n^2$
Complex	$4(5.5m - 0.5n)n^2$	$4(8m - 1.5n)n^2$	$4(13.5m - 2n)n^2$

Table 10: FLOP count for QR factorization and solution

The FLOP counts⁴ are given in Table 10 for both real and complex data arrays. Compared with the FLOP counts of the QR routines in the CMSSL library, which are $2(m - (n/3)n^2)$ and $r(4m - n)n$ for real factorization and solution respectively, the FLOP count entries of Table 10 are inflated due to the redundant operations existing in the code. One source of redundancy is the alignment in the solution routines. Operations involving the right hand side of the system are carried out on an array of shape $m \times n$, i.e., of the same shape as A , which is larger than the $m \times r$ array B . Other examples of redundancy are the multiplications in `sum(v*v, mask=mask)`, which are performed for all the elements (instead of only where the mask is true), and the multiplication by a scalar `beta` which is needed only for the first row (rather than the whole array).

The CMF version of the QR routines supports all four floating-point data types and the memory usage for each of the solution and factorization phases is given in Table 11 for real and complex data types of single- and double-precision. Note that the DPF suite only tests single-precision, real and complex data types.

Memory (in bytes)	Factorization	Solution
single, real	$24mn$	$24mn + 4m(r + 1)$
double, real	$36mn$	$44mn + 8m(r + 1)$
single, complex	$40mn$	$48mn + 8m(r + 1)$
double, complex	$68mn$	$92mn + 16m(r + 1)$

Table 11: Memory usage for QR factorization and solution

3.4 Gauss–Jordan Matrix Inversion

Given a square matrix A , the *Gauss–Jordan* routines compute the inverse matrix of A , A^{-1} , via the Gauss–Jordan elimination algorithm with partial pivoting [20, 70]. Pivoting is required if the system is not symmetric positive definite.

⁴The FLOP counts include higher order terms only.

The pivot element is chosen from the pivot row, and the columns are (in effect) permuted. At each pivoting iteration, this variant of the algorithm subtracts multiples of the pivot row from the rows above as well as below the pivot row. Thus, both the upper and lower triangular matrices are brought to zero. An analysis of the numerical behavior of the algorithm can be found in [13]. For a detailed algorithmic description see [65].

Rather than replacing the original matrix with the identity matrix, this space is used to accumulate the inverse solution. For single-precision real $n \times n$ matrices, the FLOP count is $n(n + 2 + 2n^2)$, and the memory requirement is $28n^2 + 16n$ bytes.

Since there is no alignment between the layout of 1-D arrays used as temporary arrays in swapping rows and columns of the 2-D arrays during total pivoting, data motion occurs in the swappings. *SPREAD* communication is used for spreading pivot rows and columns to 2-D temporary arrays. The following code fragment shows the communication patterns.

```

      real*8, array(n,n):: a,work
      real*8, array(n):: dumc
cmf$layout a(:news,:news)
cmf$layout dumc(:news)

C      swapping rows

      dumc=a(irow,:)
      a(irow,:)=a(icol,:)
      a(icol,:)=dumc

C      swapping columns

      dumc=a(:,irow)
      a(:,irow)=a(:,icol)
      a(:,icol)=dumc

C      spreading pivot columns

      dumc=a(:,icol)
      work=spread(a(icol,:),dim=1,ncopies=n)*spread(dumc,dim=2,ncopies=n)

```

3.5 Tridiagonal Solver

Two examples of tridiagonal solvers are provided, the first algorithm, **pcr**, uses parallel cyclic reduction, whereas the second algorithm, **conj-grad**, uses the Conjugate Gradient method.

3.5.1 Parallel Cyclic Reduction

Given a tridiagonal matrix A , represented as three 1-D arrays, and an $n \times r$ matrix B with r right hand sides, the **pcr** benchmark solves the irreducible tridiagonal system of linear equations $AX = B$ using the *parallel cyclic reduction* method [20, 26, 28], which performs the reduction and obtains the solution in one pass. Parallel implementation issues are discussed in [35, 37, 40].

The code can handle multiple instances of the system $AX = B$, specifically:

1. single instance,

2. multiple instances along one dimension,
3. multiple instances along two dimensions.

The three diagonals representing A have the same shape and are 1-D, 2-D, or 3-D arrays, for the first, second and third case respectively. One of the array dimensions is the *problem axis* [65] of extent n , i.e., the axis along which the system will be solved. The other dimensions are the *instance axes* [65] and the product of their extents is the number i of instances. For a single right hand side per instance, the array B is also 1-D, 2-D, or 3-D for the three cases respectively and has the same shape as each diagonal of A . For multiple right hand sides, B is 2-D, 3-D, or 4-D respectively. In this case, its first axis counts the right hand sides, is of extent r and is local to a processing element. Excluding the first axis, B is of the same shape as each of the arrays for the diagonal A . Although the code supports different dimensionalities for the diagonals of A , the DPF benchmark tests only the second case. Multiple right hand sides are used.

The main part of the `pcr` benchmark is shown below for single-precision real data, i instances along a single dimension and r right hand sides per instance.

```

real, array (q,1) :: a, b, c
real, array (r,q,1) :: B
cmf$ layout y(:serial,,)
real, array (q,1) :: tmp
...
b = 1.0 / b
do while (kk < size)
  a = a*b
  c = c*b
  do k=1,r
    tmp = B(k,::)
    tmp = tmp*b
    tmp = tmp -a*cshift(tmp,axis,-kk)-c*cshift(tmp,axis,kk)
    B(k,::) = tmp
  enddo
  b = one-a*cshift(c,axis,-kk)-c*cshift(a,axis,kk)
  b = 1.0 / b
  a = -a*cshift(a,axis,-kk)
  c = -c*cshift(c,axis,kk)
  kk = kk+kk
end do
...
```

The parameters `q` and `1` represent the size n of the system and the number i of instances. The parameter `axis` determines which axis is the problem axis (the other one is the instance axis). Therefore, if `axis` is 1, then $n = q$ and $i = 1$; if `axis` is 2, then $n = 1$ and $i = q$.

The communication consists of circular shifts performed by the intrinsic *CSHIFT*. The three arrays `a`, `b` and `c` of the same shape and layout are the diagonals of A . Excluding its first axis which is local, B is of the same shape and layout as each of the arrays representing A . In addition, a temporary array of the same shape and layout as the diagonals of A is used in updating the right hand side. Therefore, there is no misalignment overhead and the only communication cost is due to the *CSHIFTs*.

For i instances, the FLOP counts are $(\lceil \log n \rceil (5r + 12) + 4r - 2)ni$ and $4(\lceil \log n \rceil (5r + 12) + 4r - 2)ni$, for real and complex data types respectively. The memory usage is summarized in

Data Type	Memory (in bytes)
single, real	$4(r + 4)ni$
double, real	$8(r + 4)ni$
single, complex	$8(r + 4)ni$
double, complex	$16(r + 4)ni$

Table 12: Memory usage for parallel cyclic reduction on tridiagonal systems

Table 12 for all valid data types. The code supports all four floating-point data types, but the DPF benchmark tests only two: single-precision, real and complex.

Each element in each solution vector is updated $\log n$ times before its final value is available. However, only a few operations are performed on an element after each communication, regardless of n , and no replication or reduction is performed. The code uses *CSHIFTs* with offsets being powers of two. Tridiagonal solvers expose communication overhead to a much greater extent than dense matrix-vector multiplication. In the latter case, the number of operations per communicated element scales as $O(\sqrt{M})$, where M is the size of the local subgrid.

3.5.2 Conjugate Gradient

This benchmark (`conj-grad`) uses the Conjugate Gradient method for the solution of a single instance of a tridiagonal system. The data representation is the same as in `diff-1D`, but the solution technique is different.

The tridiagonal system is stored in three 1-D arrays. The tridiagonal matrix-vector multiplication required in the Conjugate Gradient method corresponds to a three-point stencil in one dimension. It is implemented using *CSHIFTs* as shown in the following code fragment, which also shows an inner-product of the Conjugate Gradient method.

```

      real*8, array(n)::main,low,upp,v,x,g,h
      cmf$layout main(:news),low(:news),upp(:news),x(:news),v(:news),g(:news),h(:news)

      low(1) = 0.0
      upp(n) = 0.0
      v = low*cshift(x,1,-1) + main*x + upp*cshift(x,1,1)

      anum=sum(g*h)

```

For a tridiagonal matrix of size n , the benchmark takes $15n$ FLOP counts per iteration, and uses $40n$ bytes of memory for double-precision data types. Unlike the *CSHIFTs* in the parallel cyclic reduction method, the *CSHIFTs* in the conjugate gradient method are only for nearest neighbor interaction.

3.6 Jacobi Eigenanalysis

The DPF routines are only valid for **real symmetric** matrices. Given a real symmetric matrix A of size $n \times n$, the benchmark uses the Jacobi method to compute the *eigenvalues* of the matrix A . *Eigenvectors* are *not* computed within the benchmark.

The Jacobi method makes iterative sweeps through the matrix. In each sweep, successive rotations are applied to the matrix to zero out each off-diagonal element. A sweep consists of

the application of $n(n-1)/2$ rotations. As each element is zeroed out, the elements previously zeroed out generally become non-zero again. However, with each step, the square root of the sum of the squares of the off-diagonal elements decreases, eventually approaching zero. Thus, the matrix approaches a diagonal matrix, and the diagonal elements approach the eigenvalues. For a detailed description of this method see [20] and [58]. Implementation details can be found in [65].

The main communication patterns include nearest neighbor *CSHIFTs* under masks on 1-D rotation vectors, *SPREADs* for duplicating 1-D rotation vectors into 2-D arrays, and *CSHIFTs* on 2-D arrays, as shown in the following code fragment.

```

      logical, array(n) :: even_diagonal_elements
      real, array(n) :: cos_vect, sin_vect, diag_vector, beta_vector
cmf$ layout cos_vect(:news), sin_vect(:news), even_diagonal_elements(:news)

      real, array(n,n) :: matrix, cosine, sine, other
cmf$ layout matrix(:news,:news), cosine(:news,:news), sine(:news,:news), other(:news,:news)
cmf$ layout diag_vector(:news), beta_vector(:news)

C      sending 1-D vectors to the diagonals of 2-D arrays

      forall(i=1:n) diag_vector(i) = matrix(i,i)
      forall(i=1:n-1) beta_vector(i) = matrix(i,i+1)

C      cshifts on 1-D vectors

      where(.not.even_diagonal_elements)
        cos_vect = cshift(cos_vect, dim=1, shift=-1)
        sin_vect = cshift(sin_vect, dim=1, shift=-1)
      endwhere

C      duplicate rotation vector to 2-D arrays

      cosine = spread(cos_vect, dim=1, ncopies=n)
      sine = spread(sin_vect, dim=1, ncopies=n)

c      right multiply (A*J).

      where(even_columns)
        other = -cshift(matrix,dim=2,shift=1)
      elsewhere
        other = cshift(matrix,dim=2,shift=-1)
      end where

      matrix = (matrix * cosine) + (other * sine)

```

The code exhibits a FLOP count of $6n^2 + 26n$ with a user allocated memory of $44n^2 + 28n$ and $88n^2 + 44n$ bytes for real single and double-precision matrices, respectively. The communication pattern consists of *CSHIFTs* on 2-D arrays and *SPREADs* from 1-D to 2-D arrays. The generation of the permutation matrices involves data motion between the diagonals of 2-D matrices and 1-D vectors, since there is no alignment of 1-D vectors to 2-D matrices.

The Jacobi eigenanalysis benchmark is interesting in that it uses both 1-D and 2-D arrays with an extraction of the diagonal taking place in computing rotation factors and an alignment

and broadcast taking place in applying the rotation factors. Aligning the 1-D arrays with the 2-D arrays result in poor load-balance for the computation of rotation factors, while not aligning the arrays yields good load-balance, but results in a potentially high communication cost.

3.7 Fast Fourier Transform

These routines compute the **complex-to-complex Cooley–Tukey** FFT [10]. One-, two-, or three-dimensional transforms can be carried out. An input parameter determines the axis, or combination of axes for the transform. The *forward* transform uses a *negative* twiddle exponent, and the *inverse* transform uses a *positive* twiddle exponent.

The CMF and CMSSL versions of the DPF benchmark differs with respect to twiddle factor computation. In the CMSSL version the twiddle factors are all precomputed during a *setup* phase. This enables one setup phase to be followed by multiple calls to the routines performing actual FFT computation. These respective arrays all need to have the same rank, axis extents, layout directives and data type as the array supplied in the setup routine. At the end, a *deallocate* phase deallocates the FFT setup structure and frees the parallel storage space that it required.

In the CMF version of the DPF benchmark, the twiddle computation is included in the inner loop, as indicated in the code fragment below. The CMF version implements the butterfly communication in the FFT as a sequence of *CSHIFTs* with offsets being consecutive powers of two, as shown in the following code fragment for the one-dimensional case.

```

      complex data(len), w(len), w_self(len), w_neighbor(len), data_neighbor(len)
      cmf$ layout data(:news), w(:news). w_self(:news), w_neighbor(:news), data_neighbor(:news)

C      Implement the butterfly

      do i=1,levels
        two_exp_i = 2**(i-1)
        w = ...
        where (ones .eq. 0)
          w_self = (1,0)
          w_neighbor = w
          data_neighbor = cshift(data, dim=1, shift=two_exp_i)
        elsewhere
          w_self = -w
          w_neighbor = (1,0)
          data_neighbor = cshift(data, dim=1, shift=-two_exp_i)
        end where

C Form the weighted sum of the data element and its "neighbor"

      data = data * w_self + data_neighbor * w_neighbor
    enddo

```

The structure of the code for two- and three-dimensional transforms is similar. The transform is computed in-place. The library version supports multiple-instances while the CMF version does not. However, higher dimensional versions can be clearly viewed as multiple-instance versions of lower dimensional transforms.

The DPF benchmark has the following options:

1. direction of the transform, i.e., forward or inverse along each axis,
2. ordering of each axis; normal or bit-reversed for the input and the output,
3. scaling of the transform for each axis; no scaling, scaling with the axis extent, or the square root of the axis extent.

For an axis of extent n , the FLOP counts and user allocated memory requirements at each loop iteration for the FFT routines of the various dimensions are given in Table 13. Only the leading term in the FLOP count is given. For a more detailed accounting of the number of arithmetic operations for the radix-2 Cooley–Tukey transform see, for instance, [50].

The bit-reversal operation defines a communication pattern that has many properties, studied in for example, [60] which are common with matrix transposition. They both have large demands on the network bandwidth and often cause severe network contention for many common networks and routers. With the data layout in the DPF benchmark, the bit-reversal constitutes an AAPC whenever the size of the local data set of the axis subject to bit-reversal is at least as large as the number of processing nodes along the axis subject to bit-reversal. A detailed analysis of the parallel FFT can be found in [41, 59].

FFT Dimension	FLOP Count	Memory Count	
		32-bit precision	64-bit precision
1-D FFT	$5N \log N$	$60N$	$100N$
2-D FFT	$10N^2 \log N$	$76N^2$	$116N^2$
3-D FFT	$15N^3 \log N$	$92N^3$	$136N^3$

Table 13: FLOP count and memory usage for FFT

The FFT is one of the most widely used algorithms in science, engineering design, and in signal processing. By being a very efficient algorithm, the operation count per data point is relatively low, $O(\log n)$, and communication is global and extensive. Hence, FFTs tend to expose weaknesses in communication systems, in particular a low bisection bandwidth. It is also a good benchmark for the handling of complex arithmetic, and (local) memory hierarchies.

4 Application Kernels

These benchmarks are intended to cover computations (including communication) that dominate the running time of a wide variety of scientific applications frequently implemented on scalable architectures. We characterize these benchmarks to assess their performance according to some inherent properties that inevitably dictate their computational structure and communication pattern. A preliminary classification of a similar set of applications is listed in [7].

Single node performance is at least as important as the communication related performance on most architectures. The DPF application kernels contain two “embarrassingly parallel” codes, **fermion** and **gmo**, which do not include any interprocessor communication. However, they do involve communication between the Control Processor and the processing elements in the CMF execution model. This communication is in the form of broadcast of scalar variables from the Control Processor to the processing elements.

A large number of production codes are based on structured (regular) discretizations of space, or time. Many solution methods imply data references along the axes of grids resulting from such

discretizations. For methods of relatively low order with respect to accuracy, the data references associated with updating variables at grid points are confined to neighborhoods that extends one or two grid points in all directions. For computations of this nature, the efficient support of communication as defined by the grid is often crucial. The DPF application kernels contain codes that depend on the emulation of grids distributed across processors as follows; one-dimensional grids: **diff-1D**, **diff-2D**, and **wave-1D**; two-dimensional grids: **boson**, **ellip-2D**, and **step4**; three-dimensional grids: **diff-3D**, **rp**, and **mdcell**; four-dimensional grids: **qcd-kernel**.

Since the geometries involved are often complex, unstructured (irregular) grids is the most common form of spatial discretization in engineering applications, and the finite element method is a common solution technique. The collection of elements are typically represented as a list of elements with arrays describing the connectivity or adjacency. The **fem-3D** DPF benchmark is intended to cover some of the aspects of this type of codes. This benchmark uses an iterative solver for the equilibrium equations. The execution time is dominated by matrix-vector multiplication, which is in turn dominated by *gather* and *scatter* operations on the unstructured grid (unassembled stiffness matrix).

In addition to codes solving field problems for a variety of domain shapes and media, many computations also involve discrete entities, like particles. The so-called *N-body* codes solve field equations based on particle-particle interactions with or without spatial discretization. Traditional N-body codes do not use a spatial discretization and have an arithmetic complexity of $O(N^2)$. The so-called *particle-in-cell* (pic) codes make explicit use of a grid for long-range interaction and require the representation of both particle attributes and grid-point data. Hierarchical N-body codes of arithmetic complexity $O(N \log N)$ or $O(N)$ construct a hierarchy of grids.

The DPF benchmark suite contains two codes that carry out direct interaction between all particles: **md** for particle interaction in three-dimensional space, and **n-body** for particle interaction in two-dimensional space.

The DPF codes involving both grid and particle representations are **mdcell** and **pic-gather-scatter** (with three-dimensional grids) as well as **pic-simple** (with a two-dimensional grid).

A particularly important computational kernel is the FFT, for which efficient implementations appear in many libraries. The DPF suite includes three application kernels that make extensive use of the FFT: **ks-spectral** and **pic-simple** perform FFTs on two-dimensional grids, while **wave-1D** performs FFTs on a one-dimensional grid.

The above characterization of the application kernels is summarized in Table 14. Table 15 lists the data representation and layout for the arrays used in the dominating computations in the application kernels and Table 16 summarizes the communication patterns in the codes. Table 17 tabulates the implementation techniques for the stencil, gather/scatter, and AAB communication patterns, whereas Table 18 lists the computation to communication ratio for the main loop of application codes, as well as the memory usage. The access patterns for the local axes of the arrays, their allocation mechanisms and types of array allocation are summarized in Table 19.

4.1 Boson: many-body simulation

The **boson** DPF benchmark performs quantum many-body computations for bosons on a two-dimensional lattice using a grid-based Monte Carlo technique. The algorithms are outlined and

Function	DPF code
Embarrassingly parallel	fermion, gmo
Structured grid emulation	
1-D	diff-1D, diff-2D, wave-1D
2-D	boson, ellip-2D, step4
3-D	diff-3D, rp, mdcell
4-D	qcd-kernel
Unstructured grid emulation	fem-3D
Particle interaction:	
global	
2-D	n-body
3-D	md
local	
3-D	mdcell
Particle and grid repr.	
2-D	pic-simple
3-D	pic-gather-scatter, mdcell
FFT	
1-D	wave-1D
2-D	ks-spectral, pic-simple

Table 14: Characterization of the application kernels

Code	Arrays				Unstructured grid
	1-D	2-D	3-D	4-D, 6-D, 7-D	
boson			X(:serial,::)		
diff-1D	X(:)				
diff-2D		X(:serial,:)			
diff-3D			X(:,::)		
ellip-2D		X(:,)			
fem-3D					X(:serial,::) X(:serial,serial,)
fermion			X(:,::serial,serial)		
gmo	X(:)	X(:serial,:)			
ks-spectral		X(:,)			
mdcell				X(:serial,::,::)	
md	X(:)	X(:,)			
n-body		X(:serial,:)			
pic-simple		X(:serial,:)	X(:serial,::)		
pic-gather-scatter		X(:serial,:)	X(:serial,::)		
qcd-kernel				X(:serial,::,::,::) X(:serial,serial,::,::,::)	
qmc		X(:,)		X(:serial,serial,::)	
qptransport	X(:)				
rp			X(:,::)		
step4			X(:serial,::)		
wave-1D	X(:)				

Table 15: Data representation and layout of arrays for dominating computations in the application kernels (“:serial” represents a local axis, “:” represents a parallel axis)

Communication Pattern	Arrays				Unstructured grid
	1-D	2-D	3-D	4-D, 6-D, 7-D	
Stencil	diff-1D	diff-2D	rp, diff-3D
	wave-1D	ellip-2D
		step4
Gather		pic-simple	pic-gather-scatter		fem-3D
Gather w/ combine		pic-simple			
Scatter	qptransport		pic-gather-scatter	mdcell	
Scatter w/ combine		qmc	pic-gather-scatter		fem-3D
Reduction	qptransport	ellip-2D ks-spectral md qmc rp			
Broadcast		ellip-2D gaussj md n-body rp		qmc	
AAB		md n-body			
AAPC		diff-2D			
Butterfly (FFT)	wave-1D	pic-simple ks-spectral			
Scan	qptransport	qmc	pic-gather-scatter		
CSHIFT	wave-1D	ellip-2D n-body step4	boson rp	mdcell qcd-kernel	
Send/Get		md			
Sort	qptransport	pic-gather-scatter			

Table 16: Communication patterns in application kernels

Communication Pattern	Code	Implementation Techniques
Stencil	boson ellip-2D mdcell rp wave-1D	CSHIFT
	step4	chained CSHIFT
	diff-1D diff-2D diff-3D	Array sections
Gather	fem-3D	CMSSL partitioned gather utility
	pic-gather-scatter pic-simple	FORALL w/ indirect addressing
Gather w/ combine	pic-simple	FORALL w/ SUM
Scatter	mdcell	CMF_aset_1d or FORALL w/ indirect addressing
	pic-gather-scatter	FORALL w/ indirect addressing
	qptransport	indirect addressing
Scatter w/ combine	fem-3D	CMSSL partitioned scatter utility
	pic-gather-scatter	CMF_send_add or FORALL w/ indirect addressing
	qmc	CMF_send_overwrite
AAB	md	SPREAD
	n-body	CSHIFT, SPREAD, broadcast via PM
AAPC	diff-2D	FORALL

Table 17: Implementation techniques for some common communication patterns in application kernels

demonstrated in [5, 24, 67].

The implementation uses 3-D single-precision real arrays, with the first axis representing the time axes, and the other two axes representing the two spatial axes of the grid. Nearest neighbor *CSHIFTs* are heavily used along the spatial axes. The *CSHIFTs* are interleaved with computations in a way that precludes the use of a linear stencil formulation as well as the use of *polyshift* (*PSHIFT* [19]). The main computations are scalar operations among 2-D parallel arrays, i.e., the 3-D arrays with the time axes locally (and sequentially) indexed.

For n_x , n_y , and n_t , being the lengths in the x , y and t directions respectively, and m_b being the maximum number of bosons on a site, the application takes up $20n_xn_y + 64n_t + 6000 + 2000m_b + 768n_tn_xn_y$ bytes of memory. The most computation dominant phase consumes $su4(258 + 36/n_t)n_tn_xn_y$ FLOPs for s sweeps and u updates per sweep.

This boson Monte Carlo code uses a Cartesian lattice with periodic boundary conditions and uniform site connectivity resulting in stencil communication. The implementation can benefit strongly from local optimizations.

4.2 Diffusion equation

The DPF benchmark suite includes diffusion problems in one, two, and three dimensions. A different method is used for each dimensionality of the problem. For the one-dimensional diffusion equation (**diff-1D**) a substructured tridiagonal system solver is used. For the two-dimensional case (**diff-2D**) an ADI method is used, and for the three-dimensional problem (**diff-3D**) an explicit finite difference algorithm is used.

Code	FLOP Count (per iteration)	Memory Usage (in bytes)	Communication (per iteration)
boson	$4(258 + 36/n_t)n_t n_x n_y$	s: $20n_x n_y + 64n_t + 6000$ + $2000m_b + 768n_t n_x n_y$	38 CSHIFTs
diff-1D	$13n_x + 4P \log P - 8$	d: $32n_x$	1 3-point Stencil, substructuring w/ pcr
diff-2D	$10n_x^2 - 16n_x + 16$	d: $32n_x^2$	1 3-point Stencil, 1 AAPC
diff-3D	$9(n_x - 2)(n_y - 2)(n_z - 2)$	d: $8n_x n_y n_z$	1 7-point Stencil
ellip-2D	$38n_x n_y$	d: $96n_x n_y$	4 CSHIFTs, 3 Reductions
fem-3D	$18n_v n_e$	s: $56n_v n_e + 140n_v$ + $1200n_e$	1 Gather, 1 Scatter w/combine
fermion	local matmul	d: $144n^2 + 6ln + 48p$	N/A
gmo	$6p$	s: $p \cdot (4 \cdot ns_{in} \cdot ntr_{in} +$ $4 \cdot ns_{out} \cdot (ntr_{out} + 2) +$ $8 + 12 \cdot n_{vec})$	N/A
ks-spectral	$(76 + 40 \log_2 n_x)n_x n_e$	d: $144n_x n_e$	8 1-D FFTs on 2-D arrays
mdcell	$(101 + 392n_p)n_p n_c^3$	d: $(184 + 160n_p)n_x n_y n_z$	195 CSHIFTs, 7 Scatter on local axis
md	$(23 + 51n_p)n_p$	d: $160n_p + 80n_p^2$	6 1-D to 2-D SPREADs, 3 1-D to 2-D sends, 3 2-D to 1-D Reductions
n-body			
Broadcast	$17n^2$	s: $36n$	3 Broadcasts
Broadcast w/fill	$17n^2$	s: $20n + 36m$	3 Broadcasts
SPREAD	$17n^2$	s: $36n$	3 SPREADs
SPREAD w/fill	$17n^2$	s: $20n + 36m$	3 SPREADs
CSHIFT	$17n(n - 1)$	s: $36n$	3 CSHIFTs
CSHIFT w/fill	$17n(n - 1)$	s: $20n + 36m$	3 CSHIFTs
CSHIFT w/sym.	$13.5n(n - 1) + 17n \bmod (n, 2)$	s: $48n$	3 CSHIFTs
CSHIFT w/sym.fill	$13.5n(n - 1) + 17n \bmod (n, 2)$	s: $20n + 44m$	2.5 CSHIFTs
pic-simple	$n_p + 15n_x n_y (\log n_x + \log n_y)$	d: $60n_p + 72n_x n_y$	1 Gather w/ add 1-D to 2-D, 3 FFT, 1 Gather 3-D to 2-D
pic-gather-scatter	270	s: $12n_x^3 + 88n_p$	81 Scans, 27 Scatters w/ add, 27 1-D to 3-D Scatters, 27 3-D to 1-D Gather
qcd-kernel	$606n_x n_y n_z n_t$	s: $360n_x n_y n_z n_t i$	4 CSHIFTs
qmc	$[(42 + 2n_o n_{maxw})n_p n_d n_w n_e +$ $(142n_o + 251)n_w n_e]n_b$	d: $16n_p n_d + 96n_w n_e$	n_{maxw} SPREADs 3-D to 1-D, 5 Reductions 2-D to 1-D, ($n_p n_d + 4$) Scans on 2-D, ($n_p n_d + 1$) Sends, 3 Reductions 2-D to scalar
qptransport	$34n$	d: $160n$	10 Scatters 1-D to 1-D, 1 Sort, 5 Scans, 1 CSHIFT, 1 EOSHIFT, 3 Reductions
rp	$44n_x n_y n_z$	s: $60n_x n_y n_z$	2 Reductions, 12 CSHIFTs (2 7-point Stencils)
step4	2500	s: $500n_x n_y$	128 CSHIFTs (8 16-point Stencils)
wave-1D	$29n_x + 10n_x \log n_x$	d: $64n_x$	12 CSHIFTs, 2 1-D FFTs

Table 18: Computation to communication ratio and memory usage in main loop of application kernels

Code	Local Memory Access		Array Allocation Types	
	:serial	:news	Declaration	Dummy Array
boson	1 axis: direct, strided	implicit	automatic, static	adjustable, static
diff-1D	N/A	implicit	automatic	adjustable
diff-2D	1 axis: implicit	implicit	automatic	adjustable
diff-3D	N/A	implicit	automatic	adjustable
ellip-2D	N/A	implicit	automatic	adjustable
fem-3D	2 axes: direct w/ innermost loop on leftmost axis	implicit	static	static
fermion	2 axes: direct w/ outermost loop on leftmost axis, implicit on both axes, indirect on both axes	N/A	static	static
gmo	1 axis: direct between blocks of size <code>nvec</code> , implicit in blocks, indirect	N/A	automatic, static	adjustable, static
ks-spectral	N/A	implicit	automatic, static	adjustable, static
mdcell	1 axis: indirect	implicit	automatic	adjustable
md	N/A	implicit	automatic	adjustable
n-body	1 axis: direct	implicit	automatic	adjustable
pic-simple	1 axis: direct	implicit	automatic	adjustable
pic-gather-scatter	1 axis: indirect	indirect	automatic	adjustable
qcd-kernel	2 axes: direct w/ outermost loop on leftmost axis	implicit	static	static
qmc	2 axes: direct w/ outermost loop on leftmost axis	implicit	automatic	adjustable
qptransport	N/A	implicit	automatic	adjustable
rp	N/A	implicit	automatic	adjustable
step4	1 axis: direct	implicit	static	static
wave-1D	N/A	implicit	automatic	adjustable

Table 19: Local memory access mechanisms and types of array allocation for the application kernels

All three benchmarks are examples of applications with structured Cartesian grids, solving homogeneous linear differential equations with constant boundary conditions.

4.2.1 1-D Diffusion Equation: Tridiagonal Solver

This diffusion equation simulation is the integration of the one-dimensional heat equation using the Crank-Nicholson algorithm, which is an unconditionally stable implicit method. The differencing scheme of `diff-1D` is

$$\frac{F(x, t+1) - F(x, t)}{dt} = \frac{D}{2} \left(\frac{F(x+1, t+1) - 2F(x, t+1) + F(x-1, t+1)}{dx^2} + \frac{F(x+1, t) - 2F(x, t) + F(x-1, t)}{dx^2} \right).$$

For constant temperature boundary conditions, $F(1, t) = F(n_x, t) = 0$, the resulting tridiagonal system is

$$\begin{bmatrix} 1+2g & -g & 0 & \dots & 0 \\ -g & 1+2g & -g & \dots & 0 \\ 0 & -g & 1+2g & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1+2g \end{bmatrix} \begin{bmatrix} F(1, t+1) \\ F(2, t+1) \\ F(3, t+1) \\ \vdots \\ F(n_x, t+1) \end{bmatrix} = \\
\begin{bmatrix} 1-2g & g & 0 & \dots & 0 \\ g & 1-2g & g & \dots & 0 \\ 0 & g & 1-2g & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1-2g \end{bmatrix} \begin{bmatrix} F(1, t) \\ F(2, t) \\ F(3, t) \\ \vdots \\ F(n_x, t) \end{bmatrix}$$

where $g = D \, dt / (2 \, dx^2)$ and n_x is the grid size. The right hand side of the system is formed, and, subsequently, a tridiagonal system solver is used to perform the update. This two-step process is repeated at every iteration. The total number n_t of iterations (time steps) is an input parameter. Essentially, this code is a test of the tridiagonal system solver.

The core of this benchmark is the following code fragment:

```

      real*8 F(nx), A(nx), B(nx), C(nx)
cmf$ layout F(:news), A(:news), B(:news), C(:news)
      ...
      cg = 1.0 - 2.0*g
      call tridiag_factorization(A,B,C,1)
      ...
      do (nt) times
        F(2:nx-1) = g*(F(1:nx-2) + F(3:nx)) + cg*F(2:nx-1)
        call tridiag_solver(F,A,B,C,1)
        ...
      enddo

```

CMSSL library routines are used for the tridiagonal factorization and solution. The factorization method is substructuring with parallel cyclic reduction. The domains for the substructuring are determined by the gridpoint data allocated to the memory units.

For the formation of the right hand side of the system, a centered three-point stencil must be evaluated for each grid point. The right hand side matrix is not stored explicitly. It suffices to store the scalar variable g . In the data parallel execution model of CMF, this variable is broadcast from the Control Processor to the processing elements. The communication for the stencil evaluation is expressed with parallel array sections. However, the dominant constituent of the local loop is the tridiagonal system solution.

The one-dimensional grid is of size n_x , with constant boundary conditions. The tridiagonal left-hand-side matrix is represented by three 1-D double-precision real arrays. The righthand-side matrix is not represented explicitly. The benchmark takes up $32n_x$ bytes of memory and $4(n_x - 2) + F_{sol}$ FLOPs per iteration, where F_{sol} is the number of FLOPs required for the solution part. For the CMSSL routines, substructuring with parallel cyclic reduction yields

$14n_x + 14P \log P$ operations for the factorization part and $9n_x + 4P \log P$ operations for the solution part, where P is the number of processing elements spanned by the problem axis. Thus, the total is $13n_x + 4P \log P - 8$ FLOPs per iteration, i.e., per time step.

The execution time for this benchmark is dominated by the time taken by the CMSSL routines. The substructuring is carried out as a Gauss–Jordan elimination, which only requires a single communication in order to yield a reduced size tridiagonal system with one equation per processing node [36, 40]. This system is solved using parallel cyclic reduction, which requires $\log_2 P$ communications for P processing elements [28]. For $n_x \gg P$, scalar arithmetic will dominate the performance of this code, while for $n_x \approx P$, it is very sensitive to the communication performance.

4.2.2 2-D Diffusion Equation: ADI Method

This diffusion equation simulation is the integration of the two-dimensional heat equation using the alternating direction implicit algorithm (ADI) [42, 54, 55, 69], which is an unconditionally stable implicit method. For **even** time steps, the differencing scheme of **diff-2D** is

$$\frac{F(x, y, t+1) - F(x, y, t)}{dt} = D \left(\frac{F(x+1, y, t+1) - 2F(x, y, t+1) + F(x-1, y, t+1)}{dx^2} + \frac{F(x, y+1, t) - 2F(x, y, t) + F(x, y-1, t)}{dx^2} \right).$$

For **odd** time steps, x and y are interchanged in the above equation. For constant temperature boundary conditions, $F(0, y, t) = F(n_x, y, t) = F(x, 0, t) = F(x, n_x, t) = 0$, the resulting tridiagonal system is

$$\begin{bmatrix} 1+2g & -g & 0 & \dots & 0 \\ -g & 1+2g & -g & \dots & 0 \\ 0 & -g & 1+2g & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1+2g \end{bmatrix} \begin{bmatrix} F(1, y, t+1) \\ F(2, y, t+1) \\ F(3, y, t+1) \\ \vdots \\ F(n_x, y, t+1) \end{bmatrix} = \begin{bmatrix} 1-2g & g & 0 & \dots & 0 \\ g & 1-2g & g & \dots & 0 \\ 0 & g & 1-2g & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1-2g \end{bmatrix} \begin{bmatrix} F(1, y, t) \\ F(2, y, t) \\ F(3, y, t) \\ \vdots \\ F(n_x, y, t) \end{bmatrix},$$

where $g = D \, dt / (2 \, dx^2)$, x is the problem axis, $y = 1 \dots n_x$ is the instance axis, and n_x is the length of the square grid along one axis. The right hand side of the system is formed, and, subsequently, a tridiagonal solver is used to perform the update. The array variables are laid out with one of the spatial axes local to a processing element. The tridiagonal solution is always performed along the local axis. The dependent variable $F(x, y, t)$ is transposed for the next time step. This three-step process is repeated at every iteration. The total number n_t of iterations (time steps) is an input parameter.

The lefthand-side matrix and the solution on the 2-D Cartesian grid of size $n_x \times n_x$ are implemented as four 2-D double-precision real arrays, representing the three diagonals and the solution. The right hand side matrix is not stored explicitly, but coded directly into the three-point centered difference stencil. The stencil has constant coefficients and only the scalar variable g needs to be stored for all right hand sides.

The core of this benchmark is the following code fragment:

```

      real*8 F(nx, nx), A(nx, nx), B(nx, nx), C(nx, nx)
cmf$ layout F(:serial,:news),A(:serial,:news),B(:serial,:news),C(:serial,:news)
      ...
      cg = 1.0 - 2.0*g
      call tridiag_factorization(A,B,C,1)
      ...
      do (nt) times
        F(2:nx-1, 2:nx-1) = g*(F(2:nx-1, 1:nx-2) + F(2:nx-1, 3:nx)) +
&                                cg*F(2:nx-1, 2:nx-1)
        call tridiag_solver(F,A,B,C,1)
        ...
        forall(ix=1:nx, iy=1:nx) F(ix, iy) = F(iy, ix)
      enddo

```

CMSSL library routines are used for the tridiagonal factorization and solution.

Even though the lefthand-side matrix is the same for all equations to be solved along, for instance, the x -axis, the DPF code does not exploit this fact. Thus, the benchmark code covers the general case with location dependent matrix coefficients. For the benchmark data, n_x identical systems are factored. Since the system has constant coefficients, the factorization is carried out only once. The factorization is performed with each tridiagonal system local to a processing element. Gaussian elimination is the method of choice (no pivoting is required, because the system is symmetric and diagonally dominant).

The solution, like the factorization, is always carried out along the local axis. This strategy requires a transposition of the array representing the solution variables between the solutions in the x and y directions, respectively. The transpose operation is a global-local transpose (AAPC), because the first axis is local and the other is parallel. The transpose is expressed with a *FORALL* statement.

The formation of the right hand side for each system of equations requires the evaluation of a three-point centered stencil with constant coefficients, identical to that for the single system in *diff-1D*. Array sections are used also in *diff-2D*. In the CMF execution model, g and cg are broadcast from the Control Processor to all processing elements.

For a production code with location independent matrix coefficients, this benchmark can be rewritten to reduce the work in the factorization stage by factorizing in parallel a single system matrix, as in *diff-1Dd*, since all systems share the same matrix. Moreover, the communication required for computing the right hand side can also be reduced. In the DPF code, the stencil computation is carried out along the parallel axis, but, since the axes are transposed at every time step, it could be carried out while the axis is local, by moving it to the preceding half step. The first and last time step would require special code. The only communication involved would be the global-local transpose.

The benchmark takes up $32n_x^2$ bytes of memory and $4(n_x - 2)^2 + F_{sol}$ FLOPs per iteration, where F_{sol} is the number of FLOPs required for the solution part. For the CMSSL routines,

Gaussian elimination yields $8n_x^2$ operations for the factorization part and $6n_x^2$ operations for the solution part. Thus, the total is $10n_x^2 - 16n_x + 16$ FLOPs per iteration, i.e. per time step.

Like the **diff-1D** benchmark, this benchmark is likely to be dominated by the CMSSL calls. However, the transposition is a fairly costly communication operation, and in many systems it may indeed be the dominating segment of the code with respect to execution time.

4.2.3 3-D Diffusion Equation: Explicit Finite Difference

This diffusion equation simulation is the integration of the three-dimensional heat equation using an explicit finite difference algorithm, which is stable subject to the Courant condition $dt < dx^2/D$, where D is the diffusivity. The differencing scheme of **diff-3D** is

$$\begin{aligned} \frac{F(x, y, z, t+1) - F(x, y, z, t)}{dt} = D \left(\frac{F(x+1, y, z, t) - 2F(x, y, z, t) + F(x-1, y, z, t)}{dx^2} \right. \\ + \frac{F(x, y+1, z, t) - 2F(x, y, z, t) + F(x, y-1, z, t)}{dx^2} \\ \left. + \frac{F(x, y, z+1, t) - 2F(x, y, z, t) + F(x, y, z-1, t)}{dx^2} \right) \end{aligned}$$

with constant temperature boundary conditions, $F(0, y, z, t) = F(n_x, y, z, t) = F(x, 0, z, t) = F(x, n_y, z, t) = F(x, y, 0, t) = F(x, y, n_z, t) = 0$.

The fact that the stencil coefficients are the same for all grid points (constant coefficients) is fully exploited in this DPF benchmark. Thus, neither lefthand-side nor righthand-side matrices are stored explicitly in this benchmark. Only the solution variables $F(x, y, z, t)$ are stored in array form for one time step as a 3-D double-precision array of shape $n_x \times n_y \times n_z$.

The core of this benchmark is the following code fragment:

```
      real*8 F(nx, ny, nz)
cmf$ layout F(:news, :news, :news)
      ...
      do (nt) times
        F(2:nx-1, 2:ny-1, 2:nz-1) = F(2:nx-1, 2:ny-1, 2:nz-1) +
&      c*(F(3:nx, 2:ny-1, 2:nz-1) + F(1:nx-2, 2:ny-1, 2:nz-1) +
&      F(2:nx-1, 3:ny, 2:nz-1) + F(2:nx-1, 1:ny-2, 2:nz-1) +
&      F(2:nx-1, 2:ny-1, 3:nz) + F(2:nx-1, 2:ny-1, 1:nz-2) -
&      6*F(2:nx-1, 2:ny-1, 2:nz-1))
      enddo
```

where the scalar variable c is equal to $D \, dt/dx^2$.

The operations in this benchmark are dominated by the evaluation of a seven-point centered difference stencil in three dimensions. The communication is implemented by array sections. Instead of array sections, *CSHIFTS* could have been used. This implementation technique is used in **ellip-2D** which evaluates five-point stencils in two dimensions (Section 4.3), and the **rp** benchmark which also performs a seven-point centered stencil in three dimensions (Section 4.14), just as **diff-3D**. Stencil evaluations are interesting from a compiler evaluation perspective because they occur frequently in a variety of application codes, and there are many opportunities for optimization of data motion between processing elements, and between memory and registers.

Some of these issues, related to the data parallel programming paradigm, are explored in [29], while some compiler techniques are explored in [8].

The benchmark takes up $8n_x n_y n_z$ bytes of memory and $9(n_x - 2)(n_y - 2)(n_z - 2)$ FLOPs per iteration. The total number n_t of iterations (time steps) is an input parameter.

4.3 Solution of Poisson's equation by the Conjugate Gradient method

The **ellip-2D** benchmark uses the preconditioned Conjugate Gradient method to solve Poisson's equation on a regular two-dimensional grid with Dirichlet boundary conditions. The Poisson's equation is discretized with a centered five-point stencil. The matrix-vector product in the Conjugate Gradient algorithm [12, 20, 55] takes the form of a stencil evaluation on the two-dimensional grid. In addition, one reduction and one broadcast is required for the Conjugate Gradient method.

Compared to **diff-3D** this DPF benchmark is dominated by the evaluation of a two-dimensional stencil instead of a three-dimensional stencil. Hence, for a given subgrid size, the optimum computation to communication ratio (volume/surface) is higher for **ellip-2D**. However, the inner products and broadcast require some communication, but it can be made very efficient.

The data layout and the five-point stencil are shown in the code fragment:

```

      real*8, array(nx, ny) :: ap, a0, ae, an, aw, as, p, x
cmf$ layout ap(:news, :news), a0(:news, :news), ae(:news, :news)
cmf$ layout an(:news, :news), aw(:news, :news), as(:news, :news)
cmf$ layout p(:news, :news), x(:news, :news)

c      the five-point stencil

      ap = a0*p + ae*cshift(p,1,-1) + an*cshift(p,2,-1) + aw*cshift(p,1,+1) +
&        as*cshift(p,2,+1)

c      the reduction and the broadcast

      alpha = alphan/sum(p*ap)
      x = x + alpha*p
```

The 2-D structured grids are represented by double-precision real 2-D arrays of size $n_x \times n_y$. They take up $96n_x n_y$ bytes of memory and $38n_x n_y i$ FLOPs for i iteration steps.

The three dominating operations with respect to performance on many scalable architectures in this benchmark are reduction (*SUM*), broadcast (implied in **alpha*p**), and the evaluation of the five-point centered difference stencil.

This benchmark is an example of a class of applications with structured grids, involving the solution of linear non-homogeneous differential equations with Dirichlet boundary conditions by means of an iterative solver. From a performance point of view, like **diff-3d**, this benchmark is interesting since it discloses how efficiently stencil communication and evaluation are handled by the compiler.

4.4 Finite element method in three dimensions

The **fem-3D** benchmark uses the finite element method on trilinear brick elements to solve the equilibrium equations in three dimensions. The mesh is represented as an unstructured mesh. Specifically, this code evaluates the elemental stiffness matrices and computes the displacements and the stresses at the quadrature points using the Conjugate Gradient method on unassembled stiffness matrices [33, 34]. A partitioning of the unstructured mesh is performed in an attempt to minimize the surface area (communication) for the collection of elements stored in the memory of a processing element. The partitioning is carried out by a CMSSL routine using a parallel implementation of the recursive spectral bisection technique [30, 31, 32]. The associated gather and scatter operations required for the sparse matrix–vector multiplication and assembly of residuals are also performed using CMSSL routines.

The finite element mesh is represented both as a collection of elements in a standard format [65], and as a collection of nodes:

- Each node is associated with one or more elements of the *element nodes* array, **ien**, defined by

ien(m, n) = the node number of the m th node of the n th mesh element.

This array has dimensions **ien**($nnode, nel$), where $nnode$ is the maximum number of nodes per element and nel is the number of elements in the mesh.

- The connectivity of the mesh elements across faces is represented by the *dual connectivity* array, **dual**, defined by

dual(m, n) = the element that shares face m with element n ;
dual(m, n) = 0 if face m of element n forms part of the boundary of the mesh.

This array has dimensions **dual**($nface, nel$), where $nface$ is the maximum number of faces per element and nel is the number of elements in the mesh.

The unstructured mesh is read from a file and stored in the element nodes array. The mesh is partitioned and the array of pointers derived from the unstructured mesh is reordered so that the communication required by the subsequent gather and scatter operations is reduced. The mesh partitioning and reordering of pointers are implemented via calling the CMSSL unstructured mesh partitioning utilities.

The nodes of the mesh are stored in a 2-D array whose first axis is local to a processing element. It represents the coordinates of each node. The computations of the benchmark are dominated by the sparse matrix–vector multiplication required for the Conjugate Gradient method. It is performed in three steps: a gather, a local elementwise dense matrix–vector multiplication, and a scatter. The elementwise matrix–vector multiplication exploits the fact that the elemental stiffness matrices are symmetric. The gather and scatter operations are implemented by calling the CMSSL partitioned gather/scatter utilities. The data layout and the sparse matrix–vector multiplication are shown in the following code fragment:

```

      logical y_mask(nnpe, maxel)
      integer p(nnpe, maxel), x_tmp(maxnp1)
      CMF$ layout y_mask(:serial,:), p(:serial,:), x_tmp(:)

```

```

      real y(3,maxnp1), x(3,maxnp1), xv(3,nnpe,maxel), yv(3,nnpe,maxel)
CMF$ layout y(:serial,:), x(:serial,:)
CMF$ layout xv(:serial,:serial,:), yv(:serial,:serial,:)
dimension stiffness(klen,maxel)
CMF$ layout stiffness(:serial,:)

C      call part_gather_setup(p, y_mask, x_tmp, trace, ier)
C      call part_scatter_setup(p, y_mask, x_tmp, numrep, ier)

C      1. nodes to elements -- gather
C          where (y_mask) xv = x(:,p)

      call part_vector_gather(xv, x, elements_mask, 1, 1, trace)

C      2. local matrix-vector multiply on elements
C          yv = local_matvec(stiffness, xv)
      ...

C      3. elements to nodes -- scatter with combine
C          forall (i=1:3,j=1:maxnp1,y_mask(i,j))
C      $      y(i,j) = sum(yv(i,:,:),p(:,:) .eq. j)

      call part_vector_scatter(y, yv, elements_mask, 1, 1, numrep)

```

For n_{ve} vertices per element, n_e elements overall, and n_v vertices in the domain, the application uses $56n_{ve}n_e + 140n_v + 1200n_e$ bytes of memory, and the sparse matrix–vector consumes $18n_{ve}n_e$ FLOPs.

The **fem-3D** is a good example of finite element computations on unstructured grids. In order to preserve locality of reference and allow for blocking, and the use of dense matrix optimization techniques, finite element codes are often based on unassembled stiffness matrices. Moreover, by exploiting symmetry in the elemental stiffness matrices, this code is a good test case for compiler optimizations with respect to local memory references for a somewhat more complex situation than a dense nonsymmetric matrix. Through the use of the CMSSL routines for mesh partitioning and gather and scatter operations, these operations are quite efficient and the local computations represent a significant fraction of the total execution time, except for very small submeshes.

4.5 Fermion

The **fermion** benchmark is a quantum many–body computation for fermions on a two–dimensional lattice. The solution uses a grid–based Monte Carlo method [6, 23]. The parallelization in this code is accomplished by assigning a different configuration disorder to each node. The code is “embarrassingly” parallel. Each node simulates a different physical system and there is no communication between the nodes.

The benchmark only tests one disorder and on one node. Thus, it maintains 3–D double–precision arrays, with the first axis parallel with extent four, representing the four VUs, and the second and the third axes local to each VU, representing the grid. The detailed data layout of the 3–D arrays is as follows:

```

      real*8, array(4,nxy,nxy):: tmat

```

```
cmf$ layout tmat(:,block=1:procs=4,:block=nxy:procs=1,:block=nxy:procs=1)
```

The only computation in the code is a local matrix multiplication, for which, a highly optimized C/DPEAC version is implemented via a sequence of rank-4 outer-products. Each outer-product is implemented via chained vector operations on submatrices, and the size of the submatrices is chosen to minimize the number of page faults and TLB thrashing. The timing instrumentation of this benchmark reports the total time in the outer loop, the time for local matrix multiplies (which constitutes the dominant arithmetic operations), the time for setting up the v -matrix for controlling the flip of the spins, and the time for flipping the spins.

For maximum number of flipped spins k , s sweeps in phase space, l tries in each sweep and with 2-D grid sizes $n \times n$, the benchmarks portrays a total FLOP count of $(k-2)sln^3$ and uses $144n^2 + 6ln + 48p$ bytes of memory, where p is the number of prime numbers in generating the Diophantine approximation to the new location of the flipped sites.

This benchmark falls within the class of lattice-based Monte Carlo applications and the implementation can benefit strongly from local optimization.

4.6 Seismic processing

The `gmo` benchmark is a highly optimized CMF code for a *generalized moveout* seismic kernel for all forms of Kirchhoff migration and Kirchhoff DMO (also known as $x-t$ migration and $x-t$ DMO).

The kernel code fragment:

```
      real trin(nsin*ntrin,nvus), trout(nsout*ntrout,nvus)
      real, array(nsout,nvus) :: tt, wt
      integer, array(nvus) :: kout, kin
cmf$ layout trin(:,news), trout(:,news)
cmf$ layout tt(:,news), wt(:,news)
cmf$ layout kout(news),kin(news)
      ...
      real, array(nvec,nvus) :: del
      integer, array(nvec,nvus) :: isamp, ksamp
cmf$ layout isamp(:,news), ksamp(:,news), del(:,news)
      ...
      nchunks = (jstop - jstart + 1)/nvec
      do jchunk=1,nchunks          ! stripmine this loop into vector chunks
         jbeg = (jchunk-1)*nvec

         forall(j=1:nvec,iv=1:nvus) isamp(j,iv) = tt(j + jbeg,iv)
         forall(j=1:nvec,iv=1:nvus) del(j,iv) = tt(j + jbeg,iv) - isamp(j,iv)
         forall(j=1:nvec,iv=1:nvus) isamp(j,iv) = min(isamp(j,iv),ltrace-1) + kin(iv)
         forall(j=1:nvec,iv=1:nvus) ksamp(j,iv) = j + jbeg + kout(iv)
         forall(j=1:nvec,iv=1:nvus) trout(ksamp(j,iv),iv) = trout(ksamp(j,iv),iv)
         &          + wt(j + jbeg,iv)*(trin(isamp(j,iv),iv)
         &          + del(j,iv)*(trin(isamp(j,iv) + 1,iv) - trin(isamp(j,iv),iv)))
      end do
c Cleanup for the part that does not evenly fit into the vector length
      ...
```

is carried out independently on all the processing elements (that is, the vector units), without any interprocessor communication. In order to accomplish this, the extent of the parallel axis

of the arrays is the number *nvus* of vector units on the machine.

The **gmo** kernel is optimized CMF. As such, for each vector unit, the code explicitly stripmines the main computational loop into vector chunks of length *nvec*. In order to perform the stripmining, the indices **isamp**, **ksamp** and the real variable **del** have a local axis of extent equal to the vector length *nvec*. *FORALL* statements are used to express each calculation in the loop for one whole vector chunk and across all the vector units. An outer do-loop steps through all the vector chunks sequentially.

The benchmark uses up $nvus \cdot (4 \cdot nsin \cdot ntrin + 4 \cdot nsout \cdot (ntrout + 2) + 8 + 12 \cdot nvec)$ bytes of memory and $6 \cdot i \cdot nvus$ FLOPs, where $i = jstop - jstart + 1$ is the number of iterations. The integer parameters *nvus*, *nvec*, *jstart*, *jstop*, *ltrace*, *nsin*, *nsout*, *ntrin*, *ntrout* are defined in the code.

The **gmo** benchmark is a good test case of a compiler's local memory management. The code is written explicitly with efficient local memory management in mind. The code is also quite sensitive to an efficient implementation of the *FORALL* statement. Any inadvertent call to a router and allocation of temporary variables is likely to have a significant impact on performance.

4.7 Spectral method: integration of Kuramoto–Sivashiniski equations

The **ks-spectral** benchmark uses a spectral method to integrate the Kuramoto–Sivashiniski equations on a two-dimensional regular grid. The core computation is a two-dimensional FFT. The two code versions in the DPF benchmark implement the FFT by either calling CMSSL or the CMF version of the FFT, as detailed in Section 3.7. The implementation is illustrated in the following code fragment:

```

subroutine evolution(rate, hk, frac)

  integer for(2), inv(2), nop(2), nml(2), brv(2), sca(2), nsc(2)
cmf$layout for(:serial), inv(:serial), nop(:serial), nml(:serial), brv(:serial)
cmf$layout sca(:serial), nsc(:serial)

  double precision half_lambda, frac
  double precision noise0(nx, ne), noise1(nx, ne), noise0(:news, :news)
  double precision dksq(nk, ne), hxr(nx, ne)
cmf$ layout noise1(:news, :news), dksq(:news, :news), hxr(:news, :news)
  double complex hx(nk, ne), hk(nk, ne), ik(nk, ne), rate(nk, ne)
cmf$ layout hx(:news, :news), hk(:news, :news), ik(:news, :news), rate(:news, :news)

  data for /cmssl_f_xform,      cmssl_nop    /,
c    inv /cmssl_i_xform,      cmssl_nop    /,
c    nop /cmssl_nop,          cmssl_nop    /,
c    nml /cmssl_normal,       cmssl_normal /,
c    brv /cmssl_bit_reversed, cmssl_normal /,
c    sca /cmssl_scale_n,      cmssl_noscale/,
c    nsc /cmssl_noscale,      cmssl_noscale/

  hk(1,:) = 0.0d0
  hx = ik*hk
  call fft_detailed(hx, 'ctoc', inv, brv, nml, sca, setup_id, ier)

  hxr = dble(hx)

```

```

hxr = half_lambda*hxr**2 + frac*noise1 + (1.0d0-frac)*noise0
hx = dcplx(hxr,0.0d0)
call fft_detailed(hx, 'ctoc', for, nml, brv, nsc, setup_id, ier)

rate = dksq*hk + hx
rate(1,:) = 0.0d0

```

The 2-D grids are implemented as double-precision, real, 2-D arrays of shape $n_x \times n_e$. The benchmark uses $144n_x n_e$ bytes of memory and consumes $(76 + 40 \log_2 n_x)n_x n_e i$ FLOPs for i iterations. The codes involve complex arithmetic operations in addition to the ones in the FFT subroutine. The communication pattern can only be attributed to those within the FFT subroutine which performs 1-D FFT on 2-D arrays.

4.8 Molecular dynamics, Lennard–Jones force law

Many molecular dynamics codes make use of the Lennard–Jones force law for particle interaction. With few exceptions, this force law is applied with a cut-off radius. A grid is superimposed on the domain of particles, and the cell size of the grid is chosen such that the cut-off radius is smaller than the cell size. This guarantees that interaction only involves particles in adjacent cells. Furthermore, if the cell size is only moderately larger than the cut-off radius, then, for simplicity in implementation, all particles in adjacent cells are assumed to interact and no separate interaction lists are maintained. The loss of efficiency is modest. The DPF `mdcell` code is designed for these assumptions. Accurate modeling of electrostatic forces, does however require long range interaction between particles, in which case the radius cannot be cut off. Thus, molecular dynamics codes that attempt to include electrostatic forces must evaluate interactions between all particles, i.e., such codes have many aspects in common with general N-body codes. The DPF code `md` carries out an evaluation of interactions between all particles using a Lennard–Jones potential without cut-off. The `mdcell` and the `md` codes both use a Maxwell initial distribution and a Verlet integration scheme. Both benchmarks make heavy use of parallel indirection.

4.8.1 Local forces only

In the `mdcell` code particles only interact with particles in nearby cells, thus no neighbor tables are required. The benchmark assumes periodic boundary conditions and initializes particles with the Maxwell distribution.

The method first iterates n_{eq} steps for the system to equilibrate. It then starts the main computation for n_s steps. At each step, the code performs a leap-frog update of particles via the following sequence of operations:

1. compute new forces utilizing symmetry in the force calculation,
2. move particle locations,
3. update the allocation of particles to cells according to the new particle locations. The movement is performed in two steps:
 - (a) iterate through the local axis of the cell – this is equivalent to slicing though the 3-D cells, compute whether particles at each slice need to migrate to neighbor cells and

to which neighbor cells. If migration is needed, use *CSHIFT* under mask to migrate the selected particles to a newly allocated slice of the 3-D cells. For example, if the maximal number of particles at any cell is `npart`, the newly allocated slice will be `cell(npart+1, :, :, :)`. Since cells may have different number of particles and not all particles at a slice may need to migrate, this will create bubbles in the 4-D arrays.

- (b) slice through the cells again and squeeze the bubbles via scatter operations on the local axis, i.e., within each cell.

The particles are initialized and manipulated in 4-D arrays, i.e., the first axis is local, and is used for storing particles in the same cell, while the other axes are parallel, representing the 3-D grid of cells. Utilizing the symmetry, each cell only needs to fetch directly 13 neighbor cells instead of all 26 in the neighbor-cell interaction. The fetching of neighbor cells is performed via *CSHIFT*. No linear-ordering [46] is imposed on the order of fetching the 13 neighbors, therefore the number of *CSHIFT*s is not minimized. The migration of particles in Step 3(a) and the scatter operations along local axis in Step 3(b) are shown in the following code fragment:

```

      real*8, array(cellx,celly,cellz) :: xdiff
CMF$ layout xdiff(:news,:news,:news)
      real*8, array(nmax,cellx,celly,cellz) :: x,y,z
      integer, array(nmax,cellx,celly,cellz) :: id
CMF$ layout x(:serial,:news,:news,:news),y(:serial,:news,:news,:news)
CMF$ layout z(:serial,:news,:news,:news),id(:serial,:news,:news,:news)
      integer, array(cellx,celly,cellz) :: tick,index_x,index_shift,kid
CMF$ layout tick(:news,:news,:news),index_x(:news,:news,:news)
CMF$ layout index_shift(:news,:news,:news),kid(:news,:news,:news)
      integer tick_it

C      slicing through 3-D cells, migrate under mask to new slices
C      towards + direction

      tick_it = max_part
      do i = 1, max_part

          if(coord.eq.1) then

              forall(ix=1:cellx) xtemp(ix, :, :) = x(i, ix, :, :)-ix*length_edgex

C          id records whether a particle location is empty

              where(id(i, :, :, :).eq. 0) xtemp = 0.0
              where(xtemp .gt. 0.0) index_x = 1

C          update particles moving outside region in + direction to wrap around

              where(xtemp(cellx, :, :).gt.0.0) x(i, cellx, :, :) = x(i, cellx, :, :) - edgex

          elseif (coord.eq.2) then
              ...
          endif

          if(maxval(index_x).gt.0)then
              tick_it = tick_it + 1
              xdiff = cshift(x(i, :, :, :), coord, -1)

```



```

        ydiff = cshift(y(i,:,:,),coord,-1)
        zdiff = cshift(z(i,:,:,),coord,-1)
        kid = cshift(id(i,:,:,),coord,-1)
        index_shift = cshift(index_x,coord,-1)

        where(index_shift.gt.0)
            x(tick_it,:,:,) = xdiff(:,:,)
            y(tick_it,:,:,) = ydiff(:,:,)
            z(tick_it,:,:,) = zdiff(:,:,)
            id(tick_it,:,:,) = kid(:,:,)
        tick = tick_it
        endwhere
    endif
enddo
...

C    scatter along local axis in squeezing bubbles in each cell

    max_part = maxval(tick)
    tick = 0

    do i=1,max_part
        where(id(i,:,:,) .gt. 0) tick = tick + 1

        forall(ix=1:cellx,iy=1:celly,iz=1:cellz, id(i,ix,iy,iz).gt.0)
1            x(tick(ix,iy,iz),ix,iy,iz) = x(i,ix,iy,iz)
        enddo
    enddo

```

The application uses 4-D double-precision arrays which take up $(184 + 160n_m)n_x n_y n_z$ bytes of memory, where the domain is decomposed into n_x , n_y and n_z cells along the x-, y- and z-axes respectively, and n_m is the maximum number of molecules in any cell.

Let n_{eq} be the number of steps to equilibrate, n_s be the number of simulation steps, n_c^3 be the number of computational cells, and n_{max_p} be the maximum number of particles per cell, initially set to four. Then, the total FLOP count is the sum of FLOP count of the following three stages, which are timed individually:

force calculation: $(n_{eq} + n_s)(78n_{max_p}n_c^3 + 392n_{max_p}^2n_c^3)$,

updating particle position and velocities: $(n_{eq} + n_s)23n_{max_p}n_c^3$,

adjusting particles within each cell: $7(n_{eq} + n_s)$ scatter operations on local axis.

4.8.2 Long range forces

The `md` code carries out a Lennard-Jones force law without cut-off. The interaction is evaluated directly through an $O(N^2)$ method. The all-to-all communication is implemented with a send-spread-reduce algorithm. Each particle attribute is implemented as a 1-D array. For instance, particle positions are implemented as three 1-D arrays with a location for each particle. The all-to-all communication makes use of 2-D temporary $n \times n$ arrays, where n is the number of particles. The particle information is spread across both dimensions. The DPF benchmark uses a Maxwell distribution for the molecules.

The layout of the 1-D particle arrays and the 2-D temporary arrays, as well as the send, *SPREAD*, and the reduction communication are shown in the following code fragment:

```

      real*8, array(n) :: xvel, yvel, zvel, fx, fy, fz
CMF$ layout xvel(:), yvel(:), zvel(:), fx(:), fy(:), fz(:)
      real*8, array(n,n) :: x, y, z, xt, yt, zt, fxx, fyy, fzz
CMF$ layout x(:,,:), y(:,,:), z(:,,:), xt(:,,:), yt(:,,:), zt(:,,:)
CMF$ layout fxx(:,,:), fyy(:,,:), fzz(:,,:)

C  (Since verlet scheme is used, the location and forces are computed at
C  N*dt, where the velocities computed at (N+0.5)*dt.)

C      1. update particle position -- 1--D to 2-D send. Assume force just
C  computed from previous iteration

xvel = xvel + fx*half*dt
yvel = yvel + fy*half*dt
zvel = zvel + fz*half*dt
      x(:,1) = x(:,1) + xvel*dt + fx*half*dt*dt
      y(:,1) = y(:,1) + yvel*dt + fy*half*dt*dt
      z(:,1) = z(:,1) + zvel*dt + fz*half*dt*dt
xvel = xvel + fx*half*dt
yvel = yvel + fy*half*dt
zvel = zvel + fz*half*dt

C      2. spread particle info across x-- and y-- axes
      x = spread(x(:,1), dim=2, ncopies=n)
      y = spread(y(:,1), dim=2, ncopies=n)
      z = spread(z(:,1), dim=2, ncopies=n)

      xt = spread(x(:,1), dim=1, ncopies=n)
      yt = spread(y(:,1), dim=1, ncopies=n)
      zt = spread(z(:,1), dim=1, ncopies=n)

C      3. all pairwise particle interactions
      ...

C      4. aggregate forces to the same particles -- reduction
      fx = sum(fxx,dim=2)
      fy = sum(fyy,dim=2)
      fz = sum(fzz,dim=2)

```

Including all the double-precision 1-D and 2-D arrays, the application consumes $160n + 80n^2$ bytes of memory where n is the number of molecules. The communication pattern involves *SPREADs* to implement the send-spread-reduce algorithm. This benchmark gives separate timing instrumentation for each of the following stages in the code (described for n_s iteration steps and n particles):

Verlet: updates particle positions and velocities using the Verlet integration method [27]; the FLOP count is $n_s(15n)$

Spreads: spreads particle information; no computation

PE:Forces evaluates force and potential; FLOP count is $n_s(30n^2)$

Dist:BC computes the difference between interacting particles; this is separated from potential and force evaluation because it is common to both; FLOP count is $n_s(21n^2)$

The total FLOP count is the sum of the above stages which amounts to $n_s(23n + 51n^2)$.

4.9 A generic direct N-body solver, long range forces

The DPF benchmark **n-body** consists of a suite of two-dimensional N-body solvers all of which directly compute all pairwise interactions [27]. The model problem is a two-dimensional vortex method for computing viscous, incompressible flow. The different codes in the suite allows for a comparison of different ways of programming the all-to-all communication. The different ways of programming the direct method are described in detail with code listings in [21]. The data structures consist of 2-D arrays with the first dimension local to a node, holding the particle information, and the second parallel, representing the number of particles. The all-to-all communication for the direct method is implemented with eight different communication patterns:

1. Broadcast
2. Broadcast with fill
3. *SPREAD*
4. *SPREAD* with fill
5. *CSHIFT*
6. *CSHIFT* with fill
7. *CSHIFT*, exploiting symmetry
8. *CSHIFT* with fill, exploiting symmetry

Though *SPREAD* by definition implies a broadcast of data, Broadcast above refers to an implementation of broadcast in which the Control Processor retrieves a data item to be broadcasted from the appropriate processing element, which it then broadcasts to all processing elements. The corresponding code fragment for Broadcast is given as:

```
      real*8 x(n), x_interact(n)
      cmf$layout x(:news), x_interact(:news)
```

```
      x_interact(:) = x(i)
```

On the other hand, *SPREAD* for broadcasting is coded as:

```
      real*8 x(n), x_interact(n)
      cmf$layout x(:news), x_interact(:news)
```

```
      x_interact(:) = spread(x(i), ncopies = n)
```

With the execution model of CMF, the data motion implied by the Broadcast implementation is well defined, while the definition of *SPREAD* allows for several different implementations (including the one used for Broadcast). Thus, the performance may differ significantly.

Table 18 shows the FLOP count and the memory usage for each of the eight approaches assuming n particles. The FLOP count given here does not include operations on fill-in elements for methods with *fill* communication. The memory usage is given in bytes where the underlying data type is a single-precision real, and a parameter m is defined to be $m = ((n - 1)/v + 1)v$, which represents the smallest multiple of the number of processors larger than n .

Method	FLOP Count	Memory Usage
Broadcast	$17n^2$	$36n$
Broadcast with fill	$17n^2$	$20n + 36m$
SPREAD	$17n^2$	$36n$
SPREAD with fill	$17n^2$	$20n + 36m$
CSHIFT	$17n(n - 1)$	$36n$
CSHIFT with fill	$17n(n - 1)$	$20n + 36m$
CSHIFT with exploiting symmetry	$13.5n(n - 1) + 17n \bmod (n, 2)$	$48n$
CSHIFT with exploiting symmetry with fill	$13.5n(n - 1) + 17n \bmod (n, 2)$	$20n + 44m$

Table 20: FLOP count and memory usage for vortex simulations

This benchmark is an example of a general N-body application which employs an all-to-all communication structure.

4.10 Particle-in-cell codes

The DPF benchmark suite contains two codes for particle-in-cell methods. Both codes represent a structured grid in the form of 3-D arrays, but differ in their representation of particle attributes. The `pic-simple` code implements a two-dimensional spatial grid with a third local axis for vector values at the grid points and a 2-D array for particles, with the attributes for each particle forming a local axis. The `pic-gather-scatter` code implements a three-dimensional spatial grid for a scalar field, while each particle attribute is represented by its own 1-D array.

The long-range particle interaction in particle-in-cell codes is determined by solving field equations on the grid by some efficient method, like the FFT, interpolating the field to particles in a cell, moving the particles, and then projecting back to the grid points. The `pic-simple` code scatters field data from the particles to the grid with *send-with-add*, whereas the `pic-gather-scatter` code accomplishes this task with both *send-with-add* and *sort-scan-with-add*.

4.10.1 Straightforward implementation

The `pic-simple` benchmark represents a two-dimensional electrostatic plasma simulation. It is meant to be a naive implementation of the method, representative of the sort of parallel code that users would write on their first attempt.

The particle information is represented by 2-D arrays, with the first axis being local to a processing element and storing the relevant attributes for each particle. Initially, the particles are placed randomly according to a uniform distribution inside a square box of shape $c_r \times c_r$. This box is centered in the domain. Each particle is given a random velocity in the range $[-c_v/2, c_v/2]$ along the x and y axes, which is modified with a component normal to, and proportional to, the vector from the box center to the particle. The magnitude of this modification vector is c_l . The two-dimensional grids are represented by 3-D arrays, again with the first axis being local and representing the 2-D vector field at each gridpoint.

The computations consist of scattering the charge data from the particles to the grid points using the CMF *send-with-add* utility. Charge data sent to the same gridpoint is combined on the fly, and no sophisticated interpolation scheme is used. Then, a two-dimensional FFT is used to solve Poisson's equation for the electrostatic field on the two-dimensional grid. This is followed by invoking a *gather* operation to determine the field at the particles. Then, the particle positions and velocities are advanced in time, using a leap-frog method.

The following code fragment shows the data layout and the communications described above.

```

      integer ir(nd, np)
cmf$ layout ir(:serial, :news)
      real*8 q(np), real*8 rho(nx, ny)
cmf$ layout q(:news), rho(:news, :news)
      real*8 f(nd, np), e(nd, nx, ny)
cmf$ layout f(:serial, :news), e(:serial, :news, :news)

c      particles to grid -- gather with combine

      forall(ix = 1:nx, iy = 1:ny) rho(ix, iy) = rho(ix, iy) +
&      sum(q, mask = ((ir(1, :) .eq. ix) .and. (ir(2, :) .eq. iy)))

c      grid to particles -- gather

      forall(id = 1:nd, ip = 1:np) f(id, ip) = e(id, ir(1, ip), ir(2, ip))

```

The ordering of the particles, that is, their allocation in memory has a significant impact upon the time for the gather and scatter operations. Ideally, particles would be allocated to the same memory unit as the gridpoints with which they interact. In such a case, most gather and scatter operations would be local memory operations. Moreover, the FFT computation allocating square subgrids to each memory unit allows for minimal communication and optimum utilization of the local memory hierarchy. Hence, this benchmark allows for the evaluation of several compiler and associated run-time system features, such as handling remote and local parts of gather and scatter operations separately, avoiding redundant remote communication, etc.

The FFT computation is performed via calling CMSSL FFT functions. For double-precision 2-D arrays, n_p particles and n_x and n_y extents of the x - and y -axes of the grid respectively, the application takes up $60n_p + 72n_xn_y$ bytes of memory and $n_t(n_p + 15n_xn_y(\log n_x + \log n_y))$ FLOPs for n_t simulation steps.

4.10.2 Sophisticated implementation

The **pic-gather-scatter** benchmark tests the gather/scatter operations between data structures for particles and for cells in a typical 3-D particle-in-cell application. The solver for the field equations on the grid is *not* included. Two different techniques for particle-grid interactions are used; *send-with-add* and *sort-scan-send*. The sort-scan-send version avoids congestion, which occurs in the send-with-add version. The sort-scan-send version also uses a more sophisticated interpolation function in distributing the charges (masses, etc.) to the grid.

The particle positions are stored in a 2-D array, with the first axis being local and representing the three coordinates of a particle position. The field at the gridpoints is stored in 3-D arrays, the first axis of which is local while the other two are parallel. Hence, this layout does not

minimize the surface to volume ratio.

The data layout and the communication pattern are as follows:

```

      real, array(n1, n1, n1):: rho
      real, array(npart) :: w
CMF$ layout rho(:serial, :news, :news), w(:news)

      integer, array(npart):: it1, it2, it3
      real*8, array(npart):: send_addr
cmf$ layout it1(:news), it2(:news), it3(:news), send_addr(:news)

      logical, array(npart):: cellseg
CMF$ layout cellseg(:news)

C      the scatter with add implemented using CMF utility.
C      forall (i=1:N1,j=1:N1,k=1:N1)
C      $      rho(i,j,k) = sum(w(:), mask = (it1(:) .eq. i) .and.
C      $                                          (it2(:) .eq. j) .and.
C      $                                          (it3(:) .eq. k) )

      call cmf_send_add(rho, send_addr, w, .true.)

C      the scatter after sort-scan

      forall (i = 1:npart, cellseg(i)) rho(it1(i), it2(i), it3(i)) = w(i)

```

The benchmark makes extensive use of CMF library code for many communication functions. The FLOP count of the benchmark is given separately for each functional module with a large part of it given in terms of the library functions called. The breakdown is as follows:

send-with-add version: This version for projecting the charge density of particles to grids uses 90 FLOPs, and 27 *CMF_send_add*

sort-scan-with-add version: This version for projecting the charge density of particles to grids uses 90 FLOPs, 27 *CMF_scan_adds*, and 27 1D-to-3D scatters

gathering of grid data: Gathering field data from grid points to particles uses 90 FLOPs, 27 3D-to-1D gathers, 27 *CMF_scan_copy*, 27 1D-to-1D gather operations for reordering the particles, and 27 3-D to 1-D gather operations

For an $m \times m \times m$ grid of single-precision reals and n particles, the application requires $12m^3 + 88n$ bytes of memory. The communication include a *CMF_order*, *CMF_scan_copy*, *CMF_scan_add*, *CMF_send_add*, and the gathering and scattering of data between 1-D particle arrays and 3-D grid arrays. Note that no grid solver is included in this benchmark code.

The scan-and-send sorts particles into canonical ordering using the concatenated coordinates of a particle, i.e. $(z|y|x)$, as its key. This ordering differs from the ordering of the cells and gridpoints in that the address space for the cells and gridpoints is three-dimensional. For instance, if the axes are ordered from right to left, i.e., the stride of x is one, the stride of y is equal to the length of the x -axis, etc, then the particle ordering obtained through the above sort would correspond to an ordering of cells and gridpoints where all cells in the x -direction are ordered before any cell with different y and z coordinates, and x - y -planes of cells ordered in

increasing order of z . However, by using a three-dimensional address space for the cells, cells are ordered such that cells allocated to the same processing elements are ordered consecutively in x , then y then z order before any cell in the x direction assigned to a different processing element. The cells are ordered according to a hierarchical (two-level) row or column major ordering in three dimensions. Using a coordinate-sort for the particles as described in [29] should improve the locality in the send and get operations between the 1-D particle arrays the 3-D grid arrays.

4.11 QCD kernel: staggered fermion Conjugate Gradient method

The `qcd-kernel` benchmark represents the kernel of a staggered fermion Conjugate Gradient algorithm for Quantum Chromo-Dynamics (QCD), which involves the computation of $R = [D + m]C$. The operation performed is:

$$r(x) = mc + \sum_{\mu=1}^4 [u(x)(adj)c(x + \mu) - u(x - \mu)c(x - \mu)]$$

where all the entities are complex, except m which is a real constant. r and c represent three-element “vectors” (the three colors) and u represents a 3×3 operator between colors. The aforementioned operation is carried out on a four-dimensional lattice of shape $n_x \times n_y \times n_z \times n_t$ (three spatial and one time dimension). The integer μ loops over the four dimensions of the lattice, so $x \pm \mu$ is a neighboring lattice point along dimension μ . This method is described in detail in [17], which is also an excellent source of bibliography on the subject.

The `qcd-kernel` benchmark is implemented as a multiple-instance code, with 5-D arrays for lattice data, or more for nonscalar fields. The kernel is highly optimized in the following ways:

- All the complex entities are declared as a pair of real values (real and imaginary parts separately) to avoid operations on complex data types (and relying on the compiler to translate them).
- The 3×3 operator u is represented as a vector of length nine, so that both u and its transpose can be accessed without any extra space or mapping overheads.
- Matrix operations are calculated element by element by straight-line code; the only loop overhead is the sequential loop for μ over the four dimensions.
- Only the five dimensions for the multiple-instance lattice (four dimensions for space and time and one dimension for the n_{ins} instances) are parallel; other dimensions are local. Thus, the only communication is the circular shifts $x \pm \mu$.

For n_{ins} instances of the lattice, each execution of the kernel takes up $606 \cdot n_x n_y n_z n_t n_{ins}$ FLOPs. The benchmark employs six 6-D arrays (for the real and imaginary parts of r , c and the auxiliary entity h), with their first axis local, of size $3 \cdot n_x n_y n_z n_t n_{ins}$ each. It also employs two 7-D arrays (for the real and imaginary parts of u), with their first and second axes local, of size $4 \cdot 9 \cdot n_x n_y n_z n_t n_{ins}$ each (the first dimension counts the different operators along the 4 dimensions of the lattice). All the arrays are of single-precision real type and collectively take up $360 \cdot n_x n_y n_z n_t n_{ins}$ bytes of memory. The communication pattern consists of *CSHIFTs* only, as shown in the following code fragment.

```

      real*4, array(4,9,nins,nx,ny,nz,nt):: ur,ui
      real*4, array(3,nins,nx,ny,nz,nt):: cr,ci,rr,ri,hr,hi
cmf$layout ur(:serial,:serial,:news,:news,:news,:news,:news)
cmf$layout ui(:serial,:serial,:news,:news,:news,:news,:news)
cmf$layout cr(:serial,:news,:news,:news,:news,:news)
cmf$layout ci(:serial,:news,:news,:news,:news,:news)
cmf$layout rr(:serial,:news,:news,:news,:news,:news)
cmf$layout ri(:serial,:news,:news,:news,:news,:news)
cmf$layout hr(:serial,:news,:news,:news,:news,:news)
cmf$layout hi(:serial,:news,:news,:news,:news,:news)
      ...
      rr=em*cr
      ri=em*ci
      do mu=1,4

c Second term
      ...
      hr=cshift(cr,mu+2,1)
      hi=cshift(ci,mu+2,1)
      ...
      rr(1,:,:,,:)=rr(1,:,:,,:)+
&      ur(mu,1,:,:,,:)*hr(1,:,:,,:)+ui(mu,1,:,:,,:)*hi(1,:,:,,:)+
&      ur(mu,2,:,:,,:)*hr(2,:,:,,:)+ui(mu,2,:,:,,:)*hi(2,:,:,,:)+
&      ur(mu,3,:,:,,:)*hr(3,:,:,,:)+ui(mu,3,:,:,,:)*hi(3,:,:,,:)
      ri(1,:,:,,:)=ri(1,:,:,,:)+
&      ur(mu,1,:,:,,:)*hi(1,:,:,,:)-ui(mu,1,:,:,,:)*hr(1,:,:,,:)+
&      ur(mu,2,:,:,,:)*hi(2,:,:,,:)-ui(mu,2,:,:,,:)*hr(2,:,:,,:)+
&      ur(mu,3,:,:,,:)*hi(3,:,:,,:)-ui(mu,3,:,:,,:)*hr(3,:,:,,:)
c and, similarly, compute rr(2,), ri(2,), rr(3,), ri(3,)
      ...

c Third term
      hr=0.
      hi=0.
      ...
      hr(1,:,:,,:)=hr(1,:,:,,:)-
&      ur(mu,1,:,:,,:)*cr(1,:,:,,:)+ui(mu,1,:,:,,:)*ci(1,:,:,,:)-
&      ur(mu,4,:,:,,:)*cr(2,:,:,,:)+ui(mu,4,:,:,,:)*ci(2,:,:,,:)-
&      ur(mu,7,:,:,,:)*cr(3,:,:,,:)+ui(mu,7,:,:,,:)*ci(3,:,:,,:)
      hi(1,:,:,,:)=hi(1,:,:,,:)-
&      ur(mu,1,:,:,,:)*ci(1,:,:,,:)-ui(mu,1,:,:,,:)*cr(1,:,:,,:)-
&      ur(mu,4,:,:,,:)*ci(2,:,:,,:)-ui(mu,4,:,:,,:)*cr(2,:,:,,:)-
&      ur(mu,7,:,:,,:)*ci(3,:,:,,:)-ui(mu,7,:,:,,:)*cr(3,:,:,,:)
c and, similarly, compute hr(2,), hi(2,), hr(3,), hi(3,)
      ...
      rr=rr+cshift(hr,mu+2,-1)
      ri=ri+cshift(hi,mu+2,-1)
      ...
    enddo
  ...

```

The `qcd-kernel` benchmark is interesting as an example of how efficient local arithmetic can be carried out on small matrices and it is a good reference for compiler optimizations local to a node.

4.12 Quantum Monte-Carlo

The `qmc` benchmark evaluates the ground-state energy for two fermions confined in a one-dimensional square potential well using a Green's function quantum Monte Carlo method. The basic algorithm consists of moving random walkers in configuration space (two-dimensional in this model problem) by sampling their steps from the Green's function for the Helmholtz equation, and multiplying their weights by a factor that is a function of configuration space position. For a thorough description of the algorithmic methodology see [1].

To load-balance the calculation, high-weight particles are split and low-weight particles are killed at each step. This kill-and-split algorithm is implemented with *Scans* and *sends*, and is described in [68]. The relevant code fragment follows:

```

      subroutine kill_and_split(w_new, z_new, z_initial,
&      n_particles, n_dimensions, n_walkers, n_ensemble, sum_min, count)

      integer n_particles, n_dimensions, n_walkers, n_ensemble
      integer particle, dimension, ier, ensemble, count

      real*8 z_initial(n_particles, n_dimensions)
cmf$ layout z_initial(:serial, :serial)
      real*8 z_new(n_particles, n_dimensions, n_walkers, n_ensemble)
      real*8 w_new(n_walkers, n_ensemble)
CMF$ layout z_new(:serial, :serial, :news, :news), w_new(:news, :news)
      logical send_mask(n_walkers, n_ensemble), receive_mask(n_walkers, n_ensemble)
CMF$ layout send_mask(:news, :news), receive_mask(:news, :news)
      integer n_children(n_walkers, n_ensemble)
      integer send_addr(n_walkers, n_ensemble), news_addr(n_walkers, n_ensemble)
CMF$ layout n_children(:news, :news), send_addr(:news, :news), news_addr(:news, :news)
      real*8 rtemp(n_walkers, n_ensemble), psi_g(n_walkers, n_ensemble)
CMF$ layout rtemp(:news, :news), psi_g(:news, :news)

      ....

C      determine expected number of children:

      w_new = w_new*psi_g
      n_children = 0
      call cmf_random(rtemp, 2)
      where(rtemp .lt. mod(abs(w_new), 1.0))
         n_children = int(abs(w_new)) + 1
      elsewhere
         n_children = int(abs(w_new))
      endwhere
      where(w_new .ne. 0.0) w_new = sign(1.0d0, w_new)/psi_g

C      sum scan number of children:

      call cmf_scan_add(news_addr, n_children, cmf_null, 1, cmf_upward,
1          mf_inclusive, cmf_none, .true.)

C      determine send mask, and send address:

      send_mask = .false.
```

```

    send_mask = n_children .ne. 0
    call cmf_make_send_address(send_addr)
    call cmf_deposit_grid_coordinate(geometry, send_addr, 1, news_addr, send_mask)
    forall(ensemble = 1:n_ensemble) news_addr(:, ensemble) = ensemble
    call cmf_deposit_grid_coordinate(geometry, send_addr, 2, news_addr, send_mask)

C    send and copy scan weight

    rtemp = 0.0
    call cmf_send_overwrite(rtemp, send_addr, w_new, send_mask)
    receive_mask = rtemp .ne. 0.0
    where(.not. receive_mask) rtemp = 0.0
    call cmf_scan_copy(w_new, rtemp, receive_mask, 1, cmf_downward,
1      cmf_inclusive, cmf_start_bit, .true.)

C    send and copy scan coordinates.

    do particle = 1, n_particles
      do dimension = 1, n_dimensions
        rtemp = z_initial(particle, dimension)
        call cmf_send_overwrite(rtemp, send_addr, z_new(particle, dimension, :, :),
          send_mask)
        call cmf_scan_copy(z_new(particle, dimension, :, :), rtemp, receive_mask, 1,
&      cmf_downward, cmf_inclusive, cmf_start_bit, .true.)
      enddo
    enddo

    return
  end

```

The manifestation of the fermion sign problem in this calculation is the tendency of large numbers of walkers of opposite sign to collect in the same region of configuration space. To mitigate this problem, a walker-cancellation scheme is used, as described in [1]. This amounts to an N -body problem on the walkers, and it is implemented using the sequential spread algorithm every n_c (cancellation number) iteration steps. Thus, by making n_c large, this code is a good test of the *scan-and-send* load-balancing algorithm. By making n_c small, it is a good test of the sequential *SPREAD* N -body algorithm. Both limits are examined in this test suite.

The main data structures of the code are 4-D arrays, with the first two axes being local to a node and representing a walker, and the other two axes being parallel, representing the independent ensembles of walkers. Most computations, however, are performed on 2-D arrays corresponding to the two parallel axes of the 4-D arrays. The communication includes *reductions* along one axis of these 2-D arrays, *scatter* with combine on 2-D arrays implemented via *CMF_send_overwrite*, and *scan* operations by calling the CMF library functions *CMF_scan_copy* and *CMF_scan_add* along one dimension of the 2-D arrays.

The benchmark constructs 4-D double-precision arrays representing the four axes: number of particles n_p , number of dimensions n_d , number of walkers n_w , and number of ensembles n_e . The first two axes are made local to each processor. The code also uses 2-D arrays of shape $n_w \times n_e$. All in all, the benchmark consumes $16n_p n_d + 96n_w n_e$ bytes of memory. For n_i iterations, n_b blocks, n_o permutation orderings and maximum number of walkers n_{max_w} , the FLOP count is $[(42 + 2n_o n_{max_w})n_p n_d n_w n_e + (142n_o + 251)n_w n_e]n_b n_i$.

4.13 Quadratic programming problem

The benchmark `qptransport` is an optimization code for finding optimum paths on a bipartite graph with a quadratic cost function. The benchmark generates random sparse quadratic transportation problems and solves them using an unusual alternating direction method. Each problem is based on a large, sparse, bipartite graph. The graph vertices are either “sources” (with an exogenous supply of some commodity) or “sinks” (with a demand). The edges of the graph connect the sources and sinks, and each one has a quadratic cost function. The problem is to route the commodity from the source nodes to the sink nodes so as to exactly use all of the supplies and satisfy all demands with the minimum possible total cost. The example code generates problems, solves them, and checks that the solutions seem approximately correct. However, only the solution phase is timed. It represents the graph as one long vector with many short segments of average length 16. The relevant code fragment follows:

```

real*8,array(m)  :: supply,demand,qcost,lcost,x
integer,array(m) :: src_degree,sink_degree,start_of_arc,end_of_arc
logical,array(m) :: start_src,start_sink
real*8,array(m) :: z,p,xold,zold,temp,lcoef,qcostpluslaminv,
$               width,slope,base,pi
integer,array(m) :: order

integer,parameter :: up = CMF_upward,
$               down = CMF_downward,
$               in = CMF_inclusive,
$               out = CMF_exclusive,
$               seg = CMF_segment_bit,
$               none = CMF_none

...
iter = 0
do while (viol > tol)
  iter = iter + 1
  xold = x
  zold = z
  x(end_of_arc) = z - p/lambda
  call cmf_scan_add(temp,x,start_sink,1,down,in,seg,.true.)
  call cmf_scan_copy(temp,temp,start_sink,1,up,in,seg,.true.)
  temp = x + (demand - temp)/sink_degree
  x(start_of_arc) = temp
  temp = rho*x + (1 - rho)*zold
  lcoef = lcost - p - lambda*temp
  count_out = count((lcoef < cshift(lcoef,dim=1,shift=-1)) .and.
$               (.not. start_src))

  if ((iter .eq. 1) .or. (count_out > 0)) then
    call cmf_rank(order,lcoef,start_src,1,up,seg,.true.)
    qcost(order) = qcost
    lcost(order) = lcost
    x(order) = x
    p(order) = p
    xold(order) = xold
    temp(order) = temp

```

```

        lcoef(order) = lcoef
        qcostpluslaminv(order) = qcostpluslaminv
        start_of_arc(end_of_arc) = order
        end_of_arc(order) = end_of_arc
        call cmf_scan_add(slope,qcostpluslaminv,start_src,
$                               1,up,in,seg,.true.)
    end if

    width = eoshift(lcoef,dim=1,shift=+1) - lcoef
    call cmf_scan_add(base,slope*width,start_src,1,up,out,seg,.true.)
    pi = (supply - base)/slope + lcoef
    call cmf_scan_min(pi,pi,start_src,1,down,in,seg,.true.)
    call cmf_scan_copy(pi,pi,start_src,1,up,in,seg,.true.)
    z = max(0.0d0,(pi - lcoef)*qcostpluslaminv)
    p = p + lambda*(temp - z)
    viol1 = maxval(abs(x-z))
    viol2 = maxval(abs(x-xold))
    viol3 = maxval(abs(z-zold))
    viol = max(viol1,viol2,viol3)
    if (mod(iter,printfreq) .eq. 0) then
        write(unit=*,fmt=1000) iter,viol1,viol2,viol3,count_out
    end if
end do

```

The code performs very few FLOPs, and is dominated by many rank, *Scan*, and scatter operations on 1-D arrays by calling the CMF utility functions, *CMF_rank*, *CMF_scan_add*, *CMF_scan_min*, *CMF_scan_copy*, and indirect addressing. It is believed that *rank* and *Scan* operations that could recognize and take advantage of all the short segments being short would substantially speed up this benchmark.

For double-precision vectors of size n , $160n$ bytes of memory is required and the computation kernel *sparse_transpo_adm* performs $34n$ FLOPs. The communication operations performed on the vectors are *Scans* combined with either add or copy, sends via array indexing, as well as the library sort function that performs the rank operation.

4.14 Solution of nonsymmetric linear equations using the Conjugate Gradient method

The *rp* benchmark solves a nonsymmetric linear system on a regular three-dimensional grid with Dirichlet boundary conditions. The particular method employed uses the single node version of Conjugate Gradient accelerated Cimmino row projections. That is, the Conjugate Gradient method is applied to the normal equations $A^T A x = A^T b$, where A is a nonsymmetric matrix with seven diagonals. Each unknown in the equation corresponds to a gridpoint in the three-dimensional grid, and the matrix A is represented in seven 3-D arrays.

The multiplication $A^T A x$ is implemented as two matrix-vector multiplications, i.e. $A^T(Ax)$. Since the matrix A is represented in seven 3-D arrays, each matrix-vector multiplication becomes a seven-point stencil evaluation. The stencil is implemented using *CSHIFTS*, which involve movement of data interior to the subgrid on each processor, as well as actual interprocessor communication of data on the boundary of each subgrid. The local data movement can be eliminated via restructuring the code using the array aliasing feature of CMF [29].

The main loop also includes inner products of “3-D” vectors in computing the scaling factor and the convergence error. The data layout, the seven-point stencil, and the inner-product are shown in the following code fragment:

```

      real x(mx,my,mz),xc(mx,my,mz),w(mx,my,mz),y(mx,my,mz),d(mx,my,mz)
      real xm(mx,my,mz),ym(mx,my,mz),zm(mx,my,mz)
      real xp(mx,my,mz),yp(mx,my,mz),zp(mx,my,mz)
cmf$ layout x(:news,:news,:news), xc(:news,:news,:news), w(:news,:news,:news)
cmf$ layout d(:news,:news,:news), y(:news,:news,:news)
cmf$ layout xm(:news,:news,:news), ym(:news,:news,:news), zm(:news,:news,:news)
cmf$ layout xp(:news,:news,:news), yp(:news,:news,:news), zp(:news,:news,:news)

c      perform a multiplication on x by A followed by a multiplication by A' to get y.
c      two seven-point stencils

      w = x*xc + cshift(x,3,-1)*zm + cshift(x,2,-1)*ym + cshift(x,1,-1)*xm +
&          cshift(x,1,1)*xp + cshift(x,2,1)*yp + cshift(x,3,1)*zp

      y = w*xc + cshift(w*zp,3,-1) + cshift(w*yp,2,-1) + cshift(w*xp,1,-1) +
&          cshift(w*xm,1,1) + cshift(w*ym,2,1) + cshift(w*zm,3,1)

c      inner-product and broadcast

      alpha = rho/sum(d*w)
      x = x + alpha*d

```

The grids are implemented with 3-D single-precision arrays of shape $m_x \times m_y \times m_z$ and the total arrays take up $60m_x m_y m_z$ bytes of memory. For i actual iterations, the FLOP count is $44m_x m_y m_z i$.

This code is dominated by two seven-point centered stencils in three dimensions, global reduction and broadcast. One of the two stencil evaluations has variable coefficients with a single source array, while the other stencil evaluation has a different source array for each shift. Stencils with multiple sources are common in the solution of Navier–Stokes equations.

4.15 Explicit finite difference in two dimensions

This DPF benchmark (**step4**) solves a subsonic Euler flow over a backward-facing step. The method employed uses fourth-order accurate finite difference equations. Artificial viscosity is used to stabilize the numerical method. The stencils for the x - and y -dimensions are shown in Figure 1 in matrix form. The 3's indicate that the first six stencils are of third-order accuracy, whereas the remaining stencils are sixth-order accurate. This can be verified via Taylor series expansion [55] and guarantees fourth-order overall convergence. This somewhat peculiar operator can be shown to yield a stable linear system and thus guarantee convergence of the solution [51, 52, 53, 62].

As shown in Figure 1, the maximum number of neighbors on either side of the main diagonal is four. The communication pattern for the difference operations therefore involves a 16-point stencil with four stencil points coming from each direction: north, east, south and west. The 16-point stencil is implemented as four four-point stencils, one along each axis. Each four-point stencil is in turn implemented using chained *CSHIFTS*, as shown in the following code fragment:

x_0	3	3	3	3	3			
x_1	3	3	3	3	3	3		
x_2	3	3	3	3	3	3		
x_3	3	3	3	3	3	3	3	
x_4	3	3	3	3	3	3	3	3
x_5		3	3	3	3	3	3	3
x_6				6	6	6	6	6
x_7				6	6	6	6	6
			.					
			.					

Figure 1: Matrix of stencils for x - and y - dimensions for **step4**

```

      real dest(nx,ny), scratch(nx,ny), left(4,nx,ny)
cmf$ layout dest(:news,:news), scratch(:news,:news)
cmf$ layout left(:serial,:news,:news)

      do i=1,4
        scratch = cshift (scratch,dim,-1)
        dest = dest + left(i,:,:) * scratch
      end do

```

The benchmark has a FLOP count of 700 – 800 per Runge-Kutta iteration for a total of approximately 2500 FLOPS per time step. The matrix is held in a 2-D grid of size $n_x \times n_y$ with single-precision data types, bringing the memory usage to $500n_x n_y$ bytes.

Like **diff-3D** and **rp**, **step4** is dominated by stencil computations.

4.16 Wave equation

This benchmark, **wave-1D**, simulates the inhomogeneous one-dimensional wave equation using the method of characteristics. It does a spline fit to the function at every time step, and makes heavy use of the FFT. A random number generator is used for the simulation.

The only data structure in the code is a 1-D array. Communication occurs within the computation of the FFT, and in an unbalanced four-point stencil, as shown in the following code fragment:

```

      real*8 c(nx), h(nx)
cmf$ layout c(:news), h(:news)

      c = cshift(h,1,+2) - 3.0*cschift(h,1,+1) + 3.0*h - cshift(h,1,-1)

```

Like **rp**, the benchmark provides two versions which differ in the implementation of the FFT. One version calls the CMSSL FFT, the other calls the CMF FFT code detailed in Section 3.7.

The 1-D double-precision real arrays of size n consume $64n$ bytes of memory. For i iterations, the FLOP count is $(29n + 10n \log n)i$.

5 Summary

The DPF benchmark suite is a set of data parallel Fortran codes intended for evaluating scalable parallel environments. The codes are provided in basic, optimized and several library versions. The codes contain new constructs of Fortran-90 and HPF. The benchmarks cover several aspects of the array syntax of Fortran-90 and HPF, collective communication functions, scientific software library functions, and application kernels that reflect the computational structure and communication patterns in typical scientific applications, particularly fluid dynamic simulations, fundamental physics and molecular studies in chemistry or biology. We provide performance evaluation metrics in the form of busy and elapsed times, busy and elapsed FLOP rates, and quantify performance according to the FLOP count, memory usage, communication pattern, local memory access, arithmetic efficiency as well as operation and communication counts per iteration. The benchmark suite is fully implemented in CMF and tested on the CM-5. After conversion to HPF, we expect the DPF benchmark suite to be a valuable asset in the development and evaluation of Fortran-90 and HPF compilers.

References

- [1] J.B. Anderson, C.A. Traynor, and B.M. Boghosian. Quantum chemistry by random walk: Exact treatment of many-electron systems. *Journal of Chemical Physics*, 95(10):7418–7425, November 1991.
- [2] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrisnan, and S. Weeratunga. The NAS parallel benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, Moffett Field, California, March 1994.
- [3] D. Bailey, T. Harris, W. Saphir, R. Wijngaartand, A. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, Moffett Field, California, December 1995.
- [4] David H. Bailey and J. Barton. The NAS kernel benchmark program. Technical Report NASA Technical Memorandum 86711, NASA Ames Research Center, Moffett Field, California, 1985.
- [5] G.G. Batrouni and R.T. Scalettar. *Physics Review* **B46**, 9051, 1992.
- [6] R. Blankenbecler, R.L. Sugar, and D.J. Scalapino. *Physics Review* **D24**, 2278, 1981.
- [7] Bruce Boghosian. CM-5 performance metrics suite. TMC internal memo, 1993.
- [8] Ralph G. Brickner, William George, S. Lennart Johnsson, and Alan Ruttenberg. A stencil compiler for the Connection Machine models CM-2/200. In H.J. Sips, editor, *Fourth International Workshop on Compilers for Parallel Computers*, pages 68–78. Delft Univ. of Tech., F. of Appl. Physics, 1993.
- [9] Marina C. Chen, James Cowie, and Janet Wu. Craft: A framework for F90 compiler optimization. In *5th Workshop on Compilers for Parallel Computers, Malaga, Spain, June 1995*, 1995.

- [10] James C. Cooley and J.W. Tukey. An algorithm for the machine computation of complex fourier series. *Math. Comp*, 19:291–301, 1965.
- [11] George Cybenko, Lyle Kipp, Lynn Pointer, and David Kuck. Supercomputer performance evaluation and the Perfect Benchmarks. In *1990 Proceedings of the International Conference on Supercomputing*, pages 254 – 266. IEEE Computer Society, 1990.
- [12] G. Dahlquist, Å. Björck, and N. Anderson. *Numerical Methods*. Series in Automatic Computation. Prentice Hall, Inc., Englewood Cliffs, NJ, 1974.
- [13] T.J. Dekker and W. Hoffman. Rehabilitation of the Gauss–Jordan algorithm. *Numerische Mathematik*, 54:591–599, 1989.
- [14] Jack J. Dongarra, J. R. Bunch, Cleve Moler, and G.W. Stewart. *Linpack User’s Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1979.
- [15] Alan Edelman, Steve Heller, and S. Lennart Johnsson. Index transformation algorithms in a linear algebra framework. *IEEE Trans. Parallel and Distributed Computing*, 5(11), November 1994.
- [16] M. Berry et. al. The PERFECT Club benchmarks: Effective performance evaluation of supercomputers. *International Journal of Supercomputer Applications*, 3:5 – 40, 1989.
- [17] S. Aoki et. al. *Int. Journal Mod. Physics C***2**, pages 829–947, 1991.
- [18] High Performance Fortran Forum. High Performance Fortran; language specification, version 1.0. *Scientific Programming*, 2(1 - 2):1–170, 1993.
- [19] William George, Ralph G. Brickner, and S. Lennart Johnsson. Polyshift communications software for the Connection Machine systems CM–2 and CM–200. *Scientific Programming*, 3(1):83–99, Spring 1994.
- [20] Gene Golub and Charles vanLoan. *Matrix Computations*. The Johns Hopkins University Press, second edition, 1989.
- [21] A. Greenberg, J. Mesirov, and J. Sethian. Programming direct N-body solvers on Connection Machines. Technical Report 245, Thinking Machines Corporation, Cambridge, MA, August 1992.
- [22] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc, 1990.
- [23] J.E. Hirsch. *Physics Review* **B31**, 4403, 1985.
- [24] J.E. Hirsch, R.L. Sugar, D.J. Scalapino, and R. Blankenbecler. *Physics Review* **B26**, 5033, 1982.
- [25] Roger Hockney and Michael Berry. Public international benchmarks for parallel computers: PARKBENCH committe report-1. Technical report, Netlib, Oak Ridge National Laboratory, February 1994.
- [26] Roger W. Hockney. A fast direct solution of Poisson’s equation using Fourier analysis. *J. ACM*, 12:95–113, 1965.

- [27] Roger W. Hockney and J.W. Eastwood. *Computer Simulation Using Particles*. McGraw-Hill, 1988.
- [28] Roger W. Hockney and C.R. Jesshope. *Parallel Computers 2*. Adam Hilger, 1988.
- [29] Yu Hu and S. Lennart Johnsson. Implementing $O(N)$ N-body algorithms efficiently in data parallel languages. *Journal of Scientific Programming*, 1996. Also available as TR-24-94, Harvard University, Division of Applied Sciences, September 1994.
- [30] Zdenek Johan, Kapil K Mathur, S. Lennart Johnsson, and Thomas J.R. Hughes. An efficient communication strategy for Finite Element Methods on the Connection Machine CM-5 system. *Computer Methods in Applied Mechanics and Engineering*, 113:363–387, 1994.
- [31] Zdenek Johan, Kapil K. Mathur, S. Lennart Johnsson, and Thomas J.R. Hughes. Mesh decomposition and communication procedures for finite element applications on the Connection Machine CM-5 System. In W Gentzsch and U Harms, editors, *High-Performance Computing and Networking*, volume 2, pages 233–240. Springer-Verlag, Lecture Notes in Computer Science, 1994.
- [32] Zdenek Johan, Kapil K. Mathur, S. Lennart Johnsson, and Thomas J.R. Hughes. Scalability of finite element applications on distributed-memory parallel computers. *Computer Methods in Applied Mechanics and Engineering*, 119(1–2):61–72, November 1994.
- [33] Claes Johnson. *Numerical solutions of partial differential equations by the Finite Element Method*. Cambridge University Press, 1987.
- [34] Claes Johnson and Anders Szepessy. On the convergence of a finite element method for a nonlinear hyperbolic conservation law. *Mathematics of Computation*, 49(180):427–444, 1987.
- [35] S. Lennart Johnsson. Solving narrow banded systems on ensemble architectures. *ACM TOMS*, 11(3):271–288, November 1985.
- [36] S. Lennart Johnsson. Solving tridiagonal systems on ensemble architectures. *SIAM J. Sci. Statist. Comput.*, 8(3):354–392, May 1987.
- [37] S. Lennart Johnsson. Scientific computation on large scale parallel computers. Lecture Notes for course CS 231, Harvard University, May 1995.
- [38] S. Lennart Johnsson, Tim Harris, and Kapil K. Mathur. Matrix multiplication on the Connection Machine. In *Supercomputing 89*, pages 326–332. ACM, November 1989.
- [39] S. Lennart Johnsson and Ching-Tien Ho. Spanning graphs for optimum broadcasting and personalized communication in hypercubes. *IEEE Trans. Computers*, 38(9):1249–1268, September 1989.
- [40] S. Lennart Johnsson and Ching-Tien Ho. Optimizing tridiagonal solvers for alternating direction methods on Boolean cube multiprocessors. *SIAM J. on Scientific and Statistical Computing*, 11(3):563–592, 1990.
- [41] S. Lennart Johnsson, Michel Jacquemin, and Robert L. Krawitz. Communication efficient multi-processor FFT. *Journal of Computational Physics*, 102(2):381–397, October 1992.

- [42] Jim Douglas Jr and H. Rachford. On the numerical solution of heat conduction problems in two and three space variables. *Trans. American Mathematical Society*, 82:421–439, 1956.
- [43] Kendall Square Research, Waltham, MA. *Kendall Square Research Technical Summary (of KSR-1)*, 1992.
- [44] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The DASH prototype: Logic overhead and performance. *IEEE Trans. Parallel and Distributed Systems*, 4(1):41–61, January 1993.
- [45] Woody Lichtenstein and S. Lennart Johnsson. Block cyclic dense linear algebra. *SIAM Journal of Scientific Computing*, 14(6):1257–1286, 1993.
- [46] P. S. Lomdahl, P. Tamayo, N. Gronbech-Jensen, and D. M. Beazley. 50 GFlops molecular dynamics on the Connection Machine 5. In *Supercomputing '93*, pages 520–527. IEEE, 1993.
- [47] Frank McMahon. The Livermore Fortran kernels: A test of numerical performance range. In *Performance Evaluation of Supercomputers*, pages 143 – 186. North Holland, 1988.
- [48] Message Passing Interface Forum. MPI: A message-passing interface standard. 1994.
- [49] Michael Metcalf and John Reid. *Fortran 90 Explained*. Oxford Scientific Publications, 1991.
- [50] Henri J. Nussbaumer. *Fast Fourier Transform and Convolution Algorithms*. Springer-Verlag, 1982.
- [51] Pelle Olsson. The numerical behavior of high-order finite difference methods. *Journal of Scientific Computing*, 9:445–466, 1994.
- [52] Pelle Olsson. Summation by parts, projections, and stability. i. *Mathematics of Computation*, 64:1035–1065, 1995.
- [53] Pelle Olsson. Summation by parts, projections, and stability. ii. *Mathematics of Computation*, 65, 1995. To appear.
- [54] D. W. Peaceman and H. H. Rachford Jr. The numerical solution of parabolic and elliptic differential equations. *J. SIAM*, 3:28–41, 1955.
- [55] William H. Press, P. Flannery, Saul A Teukolsky, and William Vetterling. *Numerical recipes in C: The Art of Scientific Computing*. Cambridge University Press, 1992.
- [56] Martin C. Rinard, Daniel J. Scales, and Monica S. Lam. Jade: a high-level, machine independent language for parallel programming. *Computer*, June 1993.
- [57] Daniel J. Scales and Monica S. Lam. The design and evaluation of a shared object system for distributed memory machines. In *First symposium on operating systems design and implementation*, November 1994.
- [58] Gautam Schroff and Robert Schreiber. On the convergence of the cyclic Jacobi method for parallel block orderings. Technical Report 88-11, Rensselaer Polytechnic Institute, Dept. of Computer Science, May 1988.
- [59] Nadia Shalaby. Optimal parallel fast Fourier transforms for different computational models. Technical Report TR-33-95, Harvard University, November 1995.

- [60] Nadia Shalaby and S. Lennart Johnsson. A vector space framework for parallel stable permutations. Technical Report TR-32-95, Harvard University, November 1995.
- [61] Priyamvada Sinval-Sharma, Lawrence Rauchwerger, and John Larson. Perfect benchmarks: Instrumented version. Technical Report CSRD-TR-1152, Center for Supercomputing Research and Development, University of Illinois at Urbana/Champaign, 1991.
- [62] Bo Strand. Summation by parts for finite difference approximations for d/dx . *Journal of Computational Physics*, 110:47-67, 1994.
- [63] Thinking Machines Corp. *CM-5 Technical Summary*, 1991.
- [64] Thinking Machines Corp. *CM Fortran Reference Manual, Version 2.1*, 1993.
- [65] Thinking Machines Corp. *CMSSL for CM Fortran, Version 3.1*, 1993.
- [66] Thinking Machines Corp. *DPEAC Reference Manual, CMOST Version 7.1*, 1993.
- [67] J. Tobochnik, G. G. Batrouni, and H. Gould. *Computers in Physics* **6**, 673, 1992.
- [68] C.A. Traynor, J.B. Anderson, and B.M. Boghosian. A quantum Monte Carlo calculation of the ground state energy of the hydrogen molecule. *Journal of Chemical Physics*, 94(5):3657-3664, March 1991.
- [69] E.L. Wachspress and G.J. Habetler. An alternating direction implicit iteration technique. *J. SIAM*, 8:403-424, 1960.
- [70] J.H. Wilkinson. Error analysis of direct methods of matrix inversion. *Journal of Association of Computing Machinery*, 8:281-330, 1961.